AMOS
THE CREATOR
USER GUIDE

MANDARIN
SOFTWARE

AMOS

The Creator

...by François Lionet

© Mandarin/Jawx 1990

Design and programming	François Lionet
Project manager	Richard Vanner
Manual author	Stephen Hill
Technical editor	Peter Lee
Editors	Alan McLachlan, Richard Vanner and Chris Payne
Manual typesetting by	Richard Vanner, Peter Lee and EIS

AMOS Packaging by Ellis, Ives and Sprowell Partnership, Wakefield.

Please join the AMOS Club for further technical support with AMOS. Write to: Aaron Fothergill, AMOS Club, 1 Lower Moor, Whiddon Valley, Barnstaple, North Devon, EX32 8NW.

Write to Mandarin Software for help with defective discs or other initial problems: Customer Services, Mandarin Software, Europa House, Adlington Park, Adlington, Macclesfield, Cheshire, SK10 4NP.

No material may be reproduced in whole or part without written permission from Mandarin Software. While every care has been taken to ensure this product is correct, the publishers cannot be held legally responsible for any errors or omissions in the manual or software. If you do find any, please tell us!
ISBN # 0948104961

How was it all done?

AMOS Basic was designed and programmed by François Lionet. His clever ideas and inspirational work have produced what we feel to be by far the best high-level programming language available on the Amiga to date.

AMOS was developed using the following programs:

Devpac II Assembler - HiSoft
Deluxe Paint III - Electronic Arts
Pix Mate - Progressive Peripherals & Software
Cross-Dos - Consultron
Mini Office Professional Communications - Database Software

Mandarin Software would like to thank the following people for their kind help during the development of AMOS:

Allistair Brimble, Aaron and Adam Fothergill of Shadow Software, Peter Hickman, Rico Holmes, Commodore UK for the international keyboard layouts (and the Amiga), Commodore France for the help with the A1000 problem, 17-Bit Software for samples and demos, Martyn Brown for fonts and support, Virus Free PD for Soundtracker, Simon Cook for his constructive comments and bugfinding, Lee, Alex, all other AMOS developers for their kind help and to all of you who have waited patiently for this software. We hope, like us, you feel it was well worth the wait.

This manual was written using WriteNow on the Apple Macintosh and paged up with Page Maker.

Copyright notice

Amos will enable you to create some very impressive software. It is very important that you acknowledge AMOS in your programmes using a phrase such as "Written by John Smith using AMOS," and, where possible, include the AMOS Sprite.

If your program is released commercially, the words "AMOS © 1990 Mandarin/Jawx" must be included on the back of the packaging and in the printer instructions.

Contents

1: Introduction 1
Dedication 2
Foreword 2

2: Getting started 3
Backup AMOS now 3
Installing AMOS on a single floppy system 3
Installing AMOS on a double floppy system 4
Installing AMOS onto a hard disc 4
Loading AMOS Basic 4
AMOS totorial 5
  Loading a program 5
  Deleting a program 6
  Direct mode 6
  Animation! 7
  Listing the sprite files 7
  Loading a sprite file 7
  Setting the sprite colours 8
  Displaying a sprite 8
  Animating a sprite 8
  Moving a sprite 8
  Music maestro! 9
The journey continues 9
Hints and tips 9

3: The editor 11
The menu window 11
The information line 11
The editor window 12
An introduction to direct mode 13
Loading a program 14
The AMOS file selector 14
Saving a Basic program 15
Scrolling through your files 15
Changing the current drive 15
Changing the directory 15
Setting the search path 15
Using the file selector 16
Editor tutorial 16
  Scrolling through a listing 16
  Label/procedure searches 17
  Folding a procedure definition 18
  Search/Replace 18
  Finding an item 18
  Replace 18

- Cut and paste ... 19
- Multiple programs and accessories ... 19
  - Multiple programs ... 19
  - Accessories ... 20

Direct mode ... 21

- Direct mode editor keys ... 21

The menu window ... 22

- Default menu ... 22
- The system menu ... 24
- The blocks menu ... 25
- The search menu ... 26

Keyboard macros ... 28

Conserving memory ... 30

Inside accessories ... 30

The HELP accessory ... 32

The editor control keys ... 32

- Special keys ... 33
- Editing keys ... 33
- The cursor arrows ... 33
- Program control ... 33
- Cut and Paste ... 34
- Marks ... 34
- Search/Replace ... 34
- Tabs ... 34

4: Basic principles ... 35

Variables ... 35

Types of variables ... 35

- Integers ... 35
- Real numbers ... 36
- String variables ... 36

Giving a variable a value ... 36

Arrays ... 36

Constants ... 37

Arithmetic operations ... 38

String operations ... 40

Parameters ... 41

Line numbers and labels ... 41

- Labels ... 41

Procedures ... 42

- Local and global variables ... 43
- Parameters and procedures ... 44
- Shared variables ... 45
- Returning values from a procedure ... 46
- Leaving a procedure ... 47
- Local DATA statements ... 47
- Hints and tips ... 47

Memory banks ... 48

  Types of memory bank 48
  Deleting banks 50
  Bank parameter functions 50
  Loading and saving banks 51
Memory fragmentation 52
Finding space for your variables 53

5: String functions 54
Array operations 59

6: Graphics 61
Colour 61
Line drawing commands 63
  Line types 67
Filled shapes 67
  Fill types 68
  Writing styles 70
  Advanced techniques 71

7: Control structures 73
Error handling 83

8: Text & windows 87
Text attributes 87
Cursor functions 89
Conversion functions 91
Cursor commands 92
Text input/output 96
Advanced text commands 98
Windows 99
Slider bars 104
Fonts 105
  Graphic text 105
  Installing new fonts 109
  Trouble shooting 109

9: Maths commands 111
Trigonometric functions 111
Standard mathematical functions 114
Creating random sequences 115
Manipulating numbers 116

10: Screens 119
The default screen 119
Defining a screen 119
Special screen modes 121
Extra half bright mode (EHB) 122

Hold and modify mode (HAM) ... 122
Loading a screen ... 124
Saving a screen ... 124
Moving a screen ... 125
Screen control commands ... 127
Defining the screen colours ... 131
Clearing the screen ... 131
Manipulating the contents of a screen ... 132
Scrolling the screen ... 133
Screen switching ... 134
Screen Synchronization ... 136
Special effects ... 136
Changing the copper list ... 142
Hints and tips ... 144

11: Hardware sprites ... 145
The sprite commands ... 145
Computed sprites ... 146
Creating an individual hardware sprite ... 149
The sprite palette ... 150
Controlling sprites ... 151
Conversion functions ... 153

12: Blitter objects ... 155
The bob control commands ... 161

13: Object control ... 165
The mouse pointer ... 165
Reading the joystick ... 167
Detecting collisions ... 169
    with a sprite ... 169
    with a bob ... 170
    between bobs and sprites ... 170
    with rectangular blocks ... 172
Bob priority ... 174
Miscellaneous commands ... 175

14: AMAL ... 176
AMAL principles ... 176
AMAL tutorial ... 177
    Moving an object ... 177
    Animation ... 179
    Simple loops ... 180
    Variables and expressions ... 181
        Internal registers ... 181
        External registers ... 181
        Special registers ... 182

    Operators 182
  Making decisions 183
  Generating an attack wave for a game 184
  Recording a complex movement sequence 185
AMAL commands 187
AMAL functions 191
Controlling AMAL programs from Basic 193
AMAL errors 195
  Error messages 196
Animation channels 197
  Animating a computed sprite 197
  Animating a blitter object 197
  Moving a screen 198
  Hardware scrolling 198
  Changing the screen size 199
  Rainbows 199
Advanced techniques 199
  The Autotest system 199
  Autotest commands 200
  Inside Autotest 201
Timing considerations 201
Beating the 16 object limit 202
STOS compatible animation commands 202

15: Background graphics 207
Icons 207
Screen blocks 209

16: Menus 212
Using a menu 212
Creating a simple menu 212
  Setting the title line 212
  Reading a simple menu 214
Advanced menuing features 214
  The menu hierarchy 215
  Keyboard shortcuts 219
  Menu control commands 220
Embedded menu commands 222
Alternative menu styles 227
Moveable menus 229
  Moving a menu within a program 231
  Displaying a menu at the cursor position 232

17: Sound and music 233
Simple sound effects 233
Sound channels 234
Sampled sound 235
  Creating a sample bank 237

Music .. 238
    Playing a note 240
    Waveforms and envelopes 241
Speech 246
Filter effects 248

18: The Keyboard 249
Input/Output 252

19: Other commands 254

20: Disc access 260
Drives and volumes 260
    Drives 260
    Volumes 260
    Logical devices 261
    Cross Dos 261
Directory changing 262
Common disc operations 265
Selecting a file 266
Running an AMOS program from disc 266
Checking for the existence of a file 267
Disc files 268
    Sequential files 268
    Random access files 271
The printer 274
External devices 274

21: Screen compaction 276

22: Machine level instructions 279
Number conversion 279
Memory manipulation 279
Bitwise operations 282
Using assembly language 285
Accessing the system libraries 287
Inside AMOS Basic 288

Command index 289

1: Introduction

WELCOME to the exciting world of *AMOS – The Creator!* As you know, the Amiga is a truly amazing computer. For the first time, all that power is at your fingertips.

In September 1988, Mandarin Software released STOS Basic for the ST. This made history as the first programming language to reach number one in the ST Gallup games charts! Now STOS has been rewritten from the ground up to produce AMOS Basic for the Amiga. AMOS Basic includes a vast range of over 500 commands – many of which are staggeringly powerful. You can, for instance, bounce a screen, or animate a sprite using just a single Basic instruction.

AMOS is not just another version of Basic – it's a dedicated games creation system which comes with its own built-in Animation Language (AMAL). AMAL programs are executed 50 times a second using a powerful interrupt system. They can be used to generate anything from the attack waves in an arcade game, to a silky-smooth hardware scrolling effect. At the same time, your Basic program can be doing something completely different!

Whatever your knowledge of programming, AMOS has something to offer you. If you have never written a game before, the prospect of creating your first game may be quite daunting. But do bear in mind that many of the all-time classics are uncomplicated programs with one or two original features – just look at Tetris for example. The strength of your game will depend on the quality of your ideas, and not just your programming skill. With a little help from AMOS, you'll be able to produce professional-looking games with just a fraction of the normal effort. All you really need is imagination.

If you've written a game in AMOS Basic, don't keep it to yourself. Mandarin Software is very keen to publish any program written using AMOS. Don't worry if your programming is a little rough. If your ideas are good enough, you could have a real future as a professional games writer. So please send us your programs. Mandarin would also be delighted to hear your comments or suggestions for the AMOS system. Several features in AMOS were taken directly from ideas which were sent to us from existing STOS users. Address your correspondence for the attention of Richard Vanner, Development Manager, Mandarin Software, Adlington Park, Adlington, Macclesfield SK10 4NP.

And now a word from the man who wrote this software – the amazingly talented and incredibly French, François Lionet. François started his career off as a vet, and ended up creating the original STOS Basic as a sideline.

Dedication from François Lionet

I'd like to dedicate this program to my wife Carine, for her patience in the stressed days!

Foreword

I'd like to assure you that I have put all my knowledge in this product. It's everything I've always wanted in a Basic for the Amiga.

When you program a game, you spend more than three quarters of the time writing low-level routines to display sprites, move screens and so on... This job is difficult on any computer, but it's a nightmare on the Amiga because of its power. In the BA years, (before AMOS!), making a game on the Amiga meant spending a lot of time on the utilities.

With AMOS, I've done the whole job once and for all, so all the boring bob routines, copper list calculations, disc handling and so on are done for you. You can now concentrate on the really important parts of your games – such as the scenario, the graphics and the music.

I hope AMOS will provide you with the opportunity to make your ideas turn into reality. I would really be very happy if, in a couple of years time, a hit-maker told me he had begun with AMOS!

I've designed AMOS to be flexible and extensible, so it can follow the evolution of your beloved machine. The best example of this can be found in the music routines. These are public domain and you can find the source code (in assembly language) on the AMOS program disc. Why? Because they provide you with a perfect example of how to create extensions for AMOS Basic. The same techniques can be used to add whole new instructions to AMOS, or to change the music routine to the latest 235-voice SoundTracker version 4.242 rev 1.2!

My last word will be – guess what – about piracy. Software piracy is really a crime – a slow crime. The effects can be seen with the boom of the games consoles market. Since there's no way you can copy a cartridge, all software houses are turning to them. But if you want to be able to find an affordable computer with a disc drive and a keyboard in the next few years, you'll need to support lone voices like myself. So please don't copy AMOS or give copies to your friends. You'd literally be taking the bread out of my mouth and reducing the chance of me producing the compiler and other add-ons. If you're a registered user, you'll have the maximum support and respect from us!

Well that's it. Thanks for reading. I do hope you'll find this product useful, and you'll spend lots of creative and enjoyable hours with it!

François

2: Getting started

AMOS Basic is a truly remarkable package, capable of creating games which were previously beyond your wildest dreams. All the powerful features which make the Amiga so irresistible have been incorporated into this amazing system. With the help of AMOS Basic you can develop programs which would tax the skills of even the most expert assembly language programmer.

You can for instance, effortlessly animate up to 56 hardware sprites simultaneously! This is a real achievement, especially when you consider that the Amiga's hardware only actually provides you with eight.

If you need even more action on the screen, you can use the Amiga's blitter chip as well. Blitter objects can be created in any graphics mode you like, including HAM! The only limit to the number of Blitter Objects on the screen is the amount of available memory.

Any combination of the Amiga's graphics modes can be displayed on the screen at once. Hardware scrolling isn't just possible, it's easy! There's a built-in SCREEN OFFSET command which allows you to perform the entire process directly.

In fact, the only hard part of AMOS Basic is knowing where to start! AMOS supports over 500 Basic commands, and if you've never used Basic before, you may feel a little overawed by the sheer scale of this system. When you're in unfamiliar territory, it's always useful to have a guide to show you around and point out some of the notable landmarks. That's the purpose of this chapter.

Backup AMOS now!

Before continuing however, it's vital that you back up the entire AMOS Basic package on fresh discs. This will safeguard your copy of AMOS against accidental mistakes. You'll now be able to play around with the system as much as you like, without the risk of destroying something important.

If the worst comes to the worst, we at Mandarin will be happy to replace your disc for a nominal handling charge. But you'll obviously be deprived of AMOS Basic while it's being re-duplicated.

The installation procedure varies depending on your precise set-up, but it can usually be accomplished in a matter of minutes.

Installing AMOS on a single floppy system

1. Prepare two blank discs to hold your copies of AMOS Basic.
2. Take your AMOS program disc and slide the "write-protect tab" on the right, so that you can see a small hole. This will protect your disc against unfortunate mistakes during the duplication process.
3. Start up your Amiga with your normal Workbench disc.
4. Insert your AMOS program disc into your internal drive, and select its icon with the mouse.
5. Now choose the duplicate disc option from the Workbench menu, and follow the prompts as they are displayed on the screen. If you have any problems, see the Amiga User's Manual which came with your computer. This contains a detailed explanation of the whole procedure.
6. Repeat steps 4 and 5 for the AMOS data disc.

7 Finally, place your original copies of AMOS Basic somewhere safe.

Installing AMOS on a double floppy system

1 Prepare a couple of blank discs to hold your back-ups.
2 Take your AMOS program disc and slide the "write-protect tab" on the right, so that you can see a small hole. This will protect your disc against unfortunate mistakes during the duplication process.
3 Start up your Amiga using the standard Workbench disc.
4 Insert the AMOS program disc into the internal drive, and select its icon with the mouse.
5 Put a blank disc into the second drive.
6 Drag the original AMOS disc over your new disc.
7 You will now be given the option of copying AMOS onto the new disc. Follow the prompts in the screen to back-up the disc.
8 Now repeat steps 4 to 7 for the AMOS data disc.
9 Place your original copies of AMOS Basic somewhere safe. (Preferably another room!).

Installing AMOS onto a hard disc

This requires a special installation routine on the AMOS Program disc. Here's the procedure.

1 Boot up your Amiga from your hard disc as normal.
2 Enter the CLI or SHELL.
3 Insert your AMOS program disc into the internal drive.
4 Set the directory to the internal drive with:

```
CD Df0:
```

5 You can now type the following line from the CLI prompt:

```
AMOS INSTALL.AMOS
```

6 AMOS will load into memory, and you'll be presented with the hard disc installation program. Simply follow the prompts to install AMOS straight onto your hard drive.

Loading AMOS Basic

As you might expect, AMOS Basic can be executed in a variety of different ways. You can, for instance, load AMOS directly from the Workbench by selecting its icon with the Left mouse button. Once you've entered AMOS in this way, you will be able to flick back and forth to the Workbench by pressing the Amiga and A keys from the keyboard.

In practice however, the Workbench consumes valuable memory which would be better used to hold your Basic programs. So if you're a serious user, you'll probably prefer to boot up AMOS as part of your normal start-up sequence. This will allow you to achieve the maximum possible results from the AMOS system.

To load AMOS Basic:

- Turn off your Amiga and wait for about ten seconds.
- Place a *backup* of the AMOS Program disc (Disc 1) into the internal drive.
- Now switch on your Amiga. AMOS will load into memory automatically.
- Hit a key to remove the information box and thus enter the AMOS system.

AMOS tutorial

The first thing you'll see when you enter AMOS Basic is the editor window. This is extremely easy to use, and if you've a little previous experience with computers it should be self-explanatory. Feel free to experiment as much as you like. The AMOS editor is quite intelligent, and you are unlikely to make any serious mistakes.

Now you've seen the editor window, It's time to explore some of the features that make AMOS Basic really stand out from the crowd.

Loading a program

We'll start off by showing you how you can load one of the terrific games from the AMOS data disc. We'll take the Number Leap game as an example:

- Insert the AMOS DATA disc into drive Df0: (the internal drive).

- Hold down an Amiga key on the keyboard and press the letter L. This will bring up a standard file-selector on the screen.

- Click on the disc drive label DF0 to inform AMOS that you have changed the disc.

- At the centre of the file selector there will be a list of programs which can be loaded into AMOS Basic.

- To select the Number Leap program, just position the mouse pointer over the file:

Number_Leap.AMOS

The file you have chosen will be highlighted accordingly.

- Once you've chosen your file, you can load it into the Amiga's memory by clicking twice on the left mouse button. Your game will now be entered from the AMOS DATA disc and you will be returned to the original editor screen. The contents of this window will be updated to display your new program listing.

- You can run this program by selecting the RUN button from the main menu area (or hitting F1 from the keyboard if you're feeling lazy).

The editor screen will now disappear completely and Number_Leap will be executed in front of your eyes. After you've played with this game to your satisfaction, you can exit to AMOS Basic by pressing the Ctrl and C keys simultaneously.

Ctrl+C provides an effective way of breaking into the vast majority of AMOS programs. It can be disabled from within your program using a BREAK OFF command for extra security. When the program has been broken into you can flick straight back to the editor by pressing the Spacebar key from the keyboard.

Deleting a program

Now that we've finished with the Number Leap program, we can erase it from memory with the NEW command. You won't find this option on the main menu, as it's been placed in a separate SYSTEM menu. This can be brought into view by moving the mouse pointer over the menu window and holding down the right mouse button.

To delete a program:

- Ensure the mouse pointer is over the menu area.

- Hold the Right mouse button down to bring up the SYSTEM menu.

- While the button is depressed, move the pointer over the NEW option and select it with the Left mouse key. Alternatively, you can execute this option directly from the keyboard by pressing Shift+F9.

- Type Y to confirm the operation or N to abort. (You can also highlight these options using the mouse.)

- If the current program hasn't been saved, you'll be asked whether to store it onto the disc. If you select the *Yes* option, you'll be presented with an AMOS file selector. Otherwise your program will be totally erased.

Direct mode

We'll now have a quick look at *direct* mode.This forms the centre of the AMOS Basic package and allows you to experiment with your routines and immediately observe the effects.

It's important to recognize that all the screens, sprites, and music defined in your program are completely separate from the Editor window. So no matter what you do in direct mode, you'll be able to return to your listing with just a single key-press.

- Enter direct mode by pressing Escape. The editor window will slide away and you'll be presented with the main program display.

Towards the bottom of this area will be a small screen which can be used to enter your direct mode commands. Try typing the following line, pressing *Return* to "execute":

```
Print "Your name"
```

Insert your name between the quotes to print your name on the Amiga's screen.

Now press the up and down arrows from the keyboard to move the window around the display area. As you can see, the Direct mode window is *totally* independent of the main program screen.

Animation!

So much for Direct mode. Let's experiment with some of the AMOS Basic sprite instructions. Before we can use these commands, we'll need to load a set of sprite images into memory. Stay in *direct* mode and enter the indented lines in bold as you come to them.

Listing the sprite files

We'll begin by listing all the available sprite files to the Amiga's screen.

- Ensure that the AMOS DATA disc is still in the internal drive.
- Display the disc file directory with the line:

```
Dir “AMOS_DATA:Sprites/”
```

This will display the sprite files we've supplied on the AMOS data disc. These files contain all the images which are used in the various example programs. You can create your own images using the Sprite definer accessory on the AMOS Program disc.

The Sprite definer incorporates a host of powerful drawing features which make it extremely easy to generate professional-quality animation sequences in your games.

Loading a sprite file

We can now load these sprites using the *load* command. The sprites will load into a special memory bank so don't expect to see any sprites to appear yet! Let's enter the sprites used by the Number Leap game with the following command:

```
Load “AMOS_DATA:Sprites/Frog_Sprites.abk”
```

If you make a mistake, hit F1 to get your previous line. This line can then be edited using the normal cursor keys and may be re-executed by pressing *Return*.

Now let's also load up a music file using a similar *load* command:

```
Load “AMOS_DATA:Music/Funkey.abk”
```

In order to check whether the sprites and music have been successfully loaded into memory, we'll call up the LISTBANK instruction like so:

```
Listbank
```

This prints a line like:

```
1 - Sprites    S:$0682B0  L:$000040
3 - Music      S:$043878  L:$0081FE
```

Don't worry if the numbers do not correspond as they will change depending on the available memory. The number of sprites we've just loaded can be returned directly with the LENGTH function.

```
Print Length(1)
```

```
64
```

All the way through this manual we will have lines you can type in, these lines will be highlighted in **bold**. Any text from the computer will be displayed below the program lines in plain text.

Setting the sprite colours

Each set of sprite images has its own set of colour values stored on the disc. Since these can be *very* different from your current screen colours, it's useful to be able to GRAB the colours from the sprite bank and copy them into an existing screen. This can be accomplished with the GET SPRITE PALETTE command. Enter the line:

```
Get Sprite Palette
```

All the colours in the main program screen will change immediately, but the direct mode window will be completely unaffected because it's been assigned its own separate list of colour values by the AMOS system.

Displaying a sprite

Sprites can be displayed anywhere on the screen using a simple AMOS Basic sprite command. Here's an example:

```
Sprite 8,129,50,62
```

Animating a sprite

Let's animate this object using the **AM**OS **A**nimation **L**anguage, (AMAL). AMAL is a unique animation system which can be used to move or animate your objects at incredible speed.

Note that when you're entering the following example programs, it's essential to type each line *exactly* as its appears in the listing, as otherwise you may get an unexpected syntax error.

```
Sprite 8,129,150,62
Amal 8,"Anim 0,(62,5)(63,5)(64,5);" : Amal On
```

The program above animates a small duck on the screen.Whilst it's being manipulated, the sprite can be moved around using the SPRITE command. Example:

```
Sprite 8,300,50,
```

Moving a sprite

Now for some movement!

```
Sprite 8,129,150,62 : A$="Anim 0,(62,5)(63,5)(64,5);"
A$=A$+"Loop: Move 320,0,100; Move -320,0,100; Jump Loop"
Amal 8,A$ : Amal On
```

This program animates the duck and moves it back and forth across the screen, using just three lines!

Although the instructions between the quotes may look like Basic, they're actually written in AMAL. All AMAL programs are executed 50 times a second and they can be exploited to produce silky smooth animation effects independently of your Basic programs.

Just to prove how amazing AMAL really is, hit Esc to jump back to the Basic editor. After a few moments, return to direct mode. Your Sprite will still be bouncing across the screen as if nothing had happened!

Music maestro!

For a finale, let's play the music! Ensure you're still in direct mode, turn up the volume on your monitor and start the music running with the MUSIC command like so:

```
Music 1
```

By the way, you can stop the music with the command:

```
Music Off
```

The journey continues

Hopefully, you'll now have a pretty good idea of what AMOS Basic can achieve. But so far we've only looked at a tiny fraction of AMOS Basic's power. As you experiment with the AMOS package, you'll quickly discover a whole new world, full of exciting possibilities.

AMOS Basic can't, of course, transform you into an expert games programmer overnight. Like any programming language, it does take a little time to familiarise yourself with the entire repertoire of commands. We'll therefore end this section with a few guidelines to help you on your way.

Hints and tips

- The best way to learn about AMOS is to create small programs to animate sprites, scroll screens or generate hi-score tables. Once you've gained a little confidence, you'll then be able to incorporate these routines into an actual game.

- Don't be overawed by the sheer size of the AMOS Basic language. In practice, you can achieve terrific effects with only a tiny fraction of the 500 or so commands available from AMOS. Start by mastering just a couple of instructions such as SPRITE and BOB, and then work slowly through the various sections. As you progress, you'll gradually build-up a detailed knowledge of the AMOS system.

- Although we've attempted to make this package as easy to use as possible, a thorough grounding of the general principles of Basic programming is invaluable. If you're new to Basic, you may find it helpful to purchase an introductory text such as **Alcock's Illustrating Basic. (Cambridge University Press).**

- Plan your games carefully on paper. It 's amazing how many problems can be completely avoided at the early design stages. Never attempt to tackle really large projects without prior preparation. It's the easiest way to get permanently lost.

- When you're writing a game, try to concentrate on the quality of the game play rather than the special effects. The graphics and music can always be added later if the idea's are good enough.

- Fill in the registration card and join the AMOS user club immediately! There's a regular newsletter providing an essential source of ideas, news and tips about the AMOS Basic system. You'll also have access to a growing library of public domain software, including sprites, samples, and music, all of which can be freely incorporated into your own programs.

We're expecting many exciting developments on the AMOS scene, such as extensions, utility programs and even books. So if you want to play a real part in the evolution of this package, join the AMOS Club **now!** We'll be delighted to hear from you.

3: The editor

The AMOS editor provides you with a massive range of editing facilities. Not only is it exceptionally powerful, but it's also delightfully easy to use. All commands can be executed either directly from the screen, or via an impressive range of simple keyboard alternatives. It's so friendly in fact, that If you've a little experience with computers, you'll probably be able to use it straight out of the box.

One of the most exciting features of this system, is that the listing is displayed completely separately from your main program screen. So you can instantly flick from your program display to the editor window using a single keypress (ESCape).

If you've plenty of memory, it's also possible to load several programs in AMOS Basic at a time. Each program can be edited totally independently, and it's possible to effortlessly switch between the various programs in memory by pressing just two keys from the editor.

The first thing you see after AMOS has loaded into memory is a standard credit screen. Applause applause! Press a key to remove this window and enter the editor.

The menu window

At the top of the screen, there's a menu window containing a list of the currently available commands. This forms the gateway to all AMOS Basic's powerful editing features. Commands can be quickly executed by moving the mouse pointer over an item, and hitting the left mouse button. Each command is also assigned to a particular function key.

In addition to the main menu, there are also a number of other menus. The most important of these menus is the SYSTEM menu. This can be brought into view by either holding down the right mouse button, or pressing the shift key from the keyboard.

The SYSTEM menu contains a range of important system commands such as LOAD, SAVE, NEW, etc. Like the main menu, all options can be executed using either the left mouse bottom, or by pressing an appropriate function key. Here are some examples:

F1	Run the current program
F2	Tests your program for syntax errors
Shift+F1	Load a program (Also Amiga+L)
Shift+F3	Save a program (Amiga+S)

Immediately below the menu window, there's a single line containing a range of useful information.

The information line

I L=1 **C=**1 **Text=**40000 **Chip=**91000 **Fast=**0 **Edit:**example

The markers at the far left display the editor mode (**I**nsert or **O**verwrite). There's also an indication of the **L**ine and **C**olumn you are presently editing. Alongside these markers is a list of three numbers:

TEXT: Measures the amount of memory which has been assigned to the editor window. This can be adjusted within AMOS Basic using a simple SET BUFFER command from the SEARCH MENU.

CHIP: Holds the amount of memory which can be accessed directly by the Amiga's custom chips. Don't panic if you've an unexpanded A500, and are feeling a little memory hungry. There are several ways to dramatically increase this value in your Basic programs (see the section on CONSERVING MEMORY for more details).

FAST: Lists the amount of FAST memory which has been installed in your computer. Whenever possible, AMOS will use this memory in preference to the more valuable CHIP memory.

EDIT: Displays the name of the program you are currently editing. Initially this area will be totally blank, but when you load or save a program to the disc, the new filename will be automatically entered into the information line.

The editor window

The Editor window forms the heart of the AMOS system, and allows you to type in your Basic program listings directly from the keyboard. All text is inserted at the current **cursor position**, which is indicated by a flashing horizontal line.

At the start of your session, the cursor will always be placed at the top left hand corner of the editing window. It can be moved around the current line using the left and right cursor keys.

Your line can be edited on a character by character basis using the Delete and Backspace keys. Delete erases the character immediately underneath the cursor, whereas Backspace deletes the character to the left of this cursor. As an example, type the line:

```
Print “AMOS”
```

When you press Return, your new line will be entered into AMOS Basic. Anything AMOS recognises as a command will be immediately converted into a special format.

All Basic commands begin with a Capital letter and continue in lower case. So the previous line will be displayed as:

```
Print “AMOS”
```

Similarly, all AMOS variables and procedures are displayed in CAPITALS. This lets you quickly check whether you've made a mistake in one of your program lines. Supposing for instance, you'd entered a line like:

```
Inpit “What's your name; “;name$
```

This would be displayed as:

```
Inpit “What's your name; “;NAME$
```

Since INPIT is in UPPER case, it's immediately obvious that you've made an error of some sort.

Ok, now for a little fun. Move the cursor under the Print command you entered a few moments ago and type in the following lines of Basic instructions.

```
Centre "<Touch 'n' Type Demo>
Do
   X$=Inkey$ : If X$<>"" Then Print X$;
Loop
```

Don't forget to press the Return key after each and every line. Now move the cursor through your new program using the arrow keys. Finally, press the F1 key to run this program.

The EDITOR WINDOW will disappear and a separate PROGRAM display will flip into place. The program now expects you to type in some text from the keyboard. As you can see, the program screen has its own independent cursor line. This is totally separate to the one used by the editor. So you can play about as much as you like, without changing your current editing position.

After you've finished, press Cntrl+C to abort the program. A thin line will now be displayed over the screen. This can moved using the up and down cursor arrows.

```
        Program interrupted at line 4
   >>> Loop
```

Pressing the space bar at this point would return you back to the editor. But since we've already seen the editor, let's have a brief look at the Direct mode instead. Hit the Escape key to flip this mode into place.

An introduction to Direct mode

DIRECT mode provides you with an easy way of testing your Basic programs. For the time being, we'll examine just a couple of its more interesting features.

All direct mode commands are entered into a special screen which is completely independent from the program display. You can move this screen up and down using the arrow keys from the keyboard.

At the top of the window, there's a list of 20 function key assignments. These represent a list of commands which have been previously assigned to the various function keys. They can be accessed by hitting the left or right Amiga-keys in combination with one of the various function keys

Whilst you're in direct mode, you can execute any Basic instructions you like. The only exceptions are things like loops or procedures. As with the editor, all commands should be entered into the computer by pressing the Return key. Here are some examples:

Print 42	Print a constant
ANSWER=6 : Print ANSWER*9	Perform a calculation
Curs Off	Turn off text cursor
Close Workbench	Deactivate Workbench. Saves around 40K but **aborts** multitasking operations!
Run	Run your program again.

It's important to recognise that no matter what you do in direct mode, there will be absolutely no effect on the current program listing. So you can mess about to your heart's content, with no risk of deleting something in your Basic program.

It's now time to return to the Editor window. So wave a fond farewell to Direct mode, and enter the editor by pressing Escape.

See how the cursor flashes in exactly the same position as before. This demonstrates that the two modes are completely separate. You can test this by diving back to Direct mode by hitting the Escape key once again. After you've played around with Direct mode to your satisfaction, enter the Editor window. Everything will be restored to exactly the state you left it.

Loading a program

We'll now discuss the various procedures for loading and saving your programs on the disc. As usual, these options can be executed either from the MENU window or using a range of simple two-key commands from the editor.The fastest way to load a program is to hold down either of Amiga keys, and press the letter L.

You'll now be presented with the standard AMOS file selector window. Nowadays, file selectors have become a familiar part of most packages available on the Amiga. So If you've used one before, the AMOS system will hold no real surprises. However, since the file-selector is such an integral part of AMOS Basic, it's well worth explaining it in some detail.

The AMOS file selector

Selecting a file from the disc couldn't be easier. Simply move the mouse cursor over the required filename so that it's highlighted in reversed text. To load this file into memory, click twice on the left mouse button. Alternatively, you can enter the name straight from the keyboard, and just press Return.

If you make a mistake, and wish to leave the selector without loading a file, move the mouse over the Quit button and select it with the left button. AMOS will abort your operation and display a "Not Done" message on the information line.

As an example, place your COPY of the AMOS program disc into the internal drive, and press Amiga+L to load a file. If you've been following our tutorial, AMOS will give you the option of saving the existing program first. Unless you've made any interesting changes, press "N" to enter the file-selector. Otherwise, see *Saving a program* for further instructions.

When the file selector appears, look out for a file with the name "Hithere.AMOS". Once you've found it, position the mouse over the name, and click twice on the left button. The example file will now load into memory, and the following listing will be loaded into AMOS Basic.

```
Rem Hi there AMOS users
Cls 0 : Rem Clear the screen with colour zero
Do
  Rem Get some random numbers
  X=Rnd(320):Y=Rnd(200):I=Rnd(15):P=Rnd(15)
  Ink I,P : Rem Add a little colour
  Text X,Y,"Hi there!" : Rem Graphic text
Loop
```

Move the text cursor over the text "Hi there" and insert your own message. Now press F1 to run the program. The program display will rapidly fill up with dozens of copies of your text. Press Cntrl+C to exit from this routine.

Saving a Basic program

Return to the editor window, and type Alt+S to save your current program onto the disc. If you feel like a change, hold down the right mouse key and click on the *Save As* option from the SYSTEM menu with the left button. Either way you'll jump straight back to the AMOS file selector window.

You should now enter the name of your new file straight from the keyboard. As you type, your letters will appear in a small window at the bottom of the selector. Like the editor, there's a cursor at the current typing position. This cursor can be moved around using all the normal editing keys. Finally, press Return to save your program to disc. We said it was easy...

Scrolling through your files

If your disc is reasonably full, the standard selection window won't be able to list the entire contents of your disc at once. You can page through the listing using the scroll bar to the left of the selection window.

Place the mouse over the bar, and hold down the left button.You'll now be able to drag the bar up or down with the mouse, moving the file window as you go. A similar effect can also be achieved by clicking on the arrow icons.

Changing the current drive

To the right of the file window, there's a list of drive names. The precise contents of the window will naturally depend on the devices you've connected to your Amiga.

If you have several drives, you can switch between them by simply clicking on the appropriate name. The directory of this drive will now be entered into the selection window, and the files can be chosen in the normal way.

Changing the directory

When you search through the directly listing, you'll discover several names with an asterix character (*) in front of them. These are not files at all. They are entire directories in their own right.

You can enter one of these folders by selecting them with the left mouse button. You may then chose your files directly from this folder. Note that only the files with the current extension ".AMOS" will be displayed.

Once you've opened a directory, you can set it as the default using the SETDIR button. The next time you enter the file selector or obtain a directory listing with DIR, your chosen folder will be entered automatically. Similarly, you can move back to the previous directory by clicking on the PARENT button. See PARENT for more details.

Setting the search path

Normally, AMOS will search for all filenames with the extension ".AMOS". If you want to load a file with another extension such as .BAK, you can edit the search pattern directly. This can be accomplished in the following way.

Move the text cursor to the PATH window by pressing with the up arrow from the keyboard. Now type your new path and hit Return. A full description of the required syntax can be found in the section on the DIR command.

Warning! AMOS uses its own individual search patterns which are very different from the standard Amiga Dos system. If you're unsure, delete the entire line up to the current VOLUME or DRIVE name and hit Return. This will present you with a full list of ALL the files on the present disc.

Using the file selector

Interestingly enough, it's also possible to call this file-selector directly from your own programs. For a demonstration, enter DIRECT mode (with Esc) and type the following line:

```
Print Fsel$("*.*")
```

After you've chosen a file, the name you've selected will be printed straight onto the screen! See FSEL$ for a detailed explanation of this command.

Editor tutorial

We'll now have a brief look at some of the more advanced editing features available from the AMOS editor. We'll start by loading an example program from the disc. Just for a challenge, we've placed this in a separate MANUAL folder on the AMOS Program disc.

Insert your COPY of the program disc into your Amiga's internal drive and call up the file-selector window with Amiga+L. Now open the MANUAL folder by selecting it with the left button. A new list of folders will be displayed in the file-selection window. As you can see, there's one folder for each chapter in this manual.

To list the files available for the current chapter, open the "Chapter_3" folder. Simply place the cursor over this name, and click on the left button. You'll now be presented with all the examples files for this chapter.

Finally, load **Example 3.1.AMOS** into memory by selecting it with the mouse. After a few moments the program will load into AMOS Basic. In case you're wondering, it's a small program to display a working dialogue box on the screen. Hit F1 for a quick demonstration. When you click on one of the buttons, the program simply displays its number and exits back to Basic.

If you examine this program, you should discover a couple of important facts. Firstly, there's not a line number in sight. Due to the power of AMOS Basic, line numbers aren't really needed. So we've made them completely optional in your Basic programs. It's entirely up to you whether you wish to use them.

Another interesting feature is that there seems to be a lot of lines starting with a strange "Procedure" statement. These form the starting points for all the procedure definitions in this program.

Procedures are independent program sections with their own lists of variables and data statements. They are rather similar to the GOSUBs you'd find in standard Basic. However, they are **much** more powerful! We'll be discussing them in detail in Chapter 4.

You can examine this program in several different ways:

Scrolling through a listing

Alongside the main editor window are two "scroll bars". These allow you to page through your listing with the mouse.

Move the mouse pointer over the Vertical bar and hold down the left button. Now drag the bar down the screen. The editor window will scroll smoothly downwards through the listing. You can also scroll the program using the Arrow Icons at the top and bottom of this bar. Clicking on these icons moves the line exactly one place in the required direction.

At the far bottom of the editor window, there's a horizontal scroll bar. This can be used to move the window left and right in exactly the same way.

If you prefer to use the keyboard for your editing, you'll be pleased to discover that there are dozens of equivalent keyboard options as well.Try the following:

• Up Arrow (From keyboard)	Moves window up by a single line.
• Down	Scrolls window down by a line.
• Cntrl+up Arrow	Shift the listing to the previous page.
• Cntrl+down Arrow	Moves the listing to the next page.

All the keyboard options obey the same basic principles. So once you've familiarised yourself with one command, the rest are easy. A full list of these commands can be found towards the end of this chapter.

Now we've looked at the program, it's time to actually change something. Search through the program listing until your find the line:

```
ALERT[50,”Alert box”,””,”Ok”,”Cancel”,1,2]
```

This calls a Basic procedure which displays a working alert box on the screen. The format of this procedure is:

```
ALERT[Y coord,Title1$,Title2$,Button1$,Button$,Paper, Ink]
```

Let's change this alert to something a little more exciting. Move the cursor over the above statement, and edit the line with the cursor keys so that it looks like so:

```
ALERT[50,”Exterminate!”,”Stephen”,”Yep!”,”Nope!”,13]
```

Execute the program by pressing F1 or selecting RUN from the main menu. You'll be given the unique option of stopping the manual author in his tracks. Select a button with the mouse and make you choice. Ouch! That hurt!

In practice, you can change the title and the buttons to literally anything you like. Feel free to use this routine in your own programs.

Hopefully, the above example will have provided you with a real spur to use procedures in your own programs. In order to aid you in this task, we've built a powerful range of special editing features into the AMOS editor.

Label/procedure searches

If your program is very long, it can be quite hard to find the starting points of your various procedure definitions. We've therefore included the ability to jump straight to the next procedure definition in your program, using just two keys (Alt+Arrow).

For an example of this feature, place the cursor at the start of the listing and press Alt+down arrow. Your cursor will be immediately moved to the beginning of the first procedure definition in the current program (ALERT). You can repeat this process to jump to each procedure definition in turn. Once you've reached the end of the listing, you can jump upwards through the listing with Alt+up arrow in an identical way.

This system is not just limited to procedures of course. It also works equally well with Labels or line numbers. So even if you don't need procedures, you'll still find a use for this feature.

Folding a procedure definition

If you build up your programs out of a list of frequently used procedures, your listings can easily be cluttered with the definitions of all your various library routines.

Fortunately, help is at hand. With a simple call to the Fold command, you can hide away any of your procedure definitions from your listings. These routines can be used in your program as normal, but their definitions will be replaced by a single Procedure statement. Example:

Position the cursor anywhere in the definition of ALERT and click on the *Fold/Unfold* option from the menu window. Bing! The contents of your procedure will vanish into thin air! Despite this, you can run the program with no ill effects. The only change has been in the appearance of the listing in the editor window.

If you want to modify this procedure, it's easy enough to get back to the original listing. Just select *Fold/Unfold* again, and your procedure will be expanded to it's full glory.

It's also possible to fold ALL the procedure in your program at once. This uses an option on the SEARCH menu called *Close All*. To bring the Search menu onto the screen, click on the button with the same name, or press F5 from the keyboard. Now select the *Close All* button to remove the procedure definitions from the current program.

The effect on **EXAMPLE 3.1** is dramatic! The entire program now fits into just a single screen. So you can instantly see the procedures we've been using in the program. Each procedure definition can be edited individually by expanding it with the *Fold/Unfold* button. Or you can unfold the the whole program with *Open All* from the search menu.

Search/Replace

The search/replace commands provided by the AMOS Basic editor are accessed through a special SEARCH menu which can be called up either from the menu window or by pressing function key F4.

Finding an item

We will continue our tutorial with a brief look at of some of the Search/Replace instructions. Let's start with the FIND command .

This can be executed either directly from the SEARCH menu or using the keys Cntrl+F from the keyboard. When you select this command, you will be asked to enter the search string.

For an example, press Cntrl+F and type *Rem* at the prompt. AMOS will now search for the next *Rem* statement in your program, starting from the current cursor position. If the search is successful, the cursor will be placed over the requested item.

The search can now be repeated from this point with the *Find Next* option (Cntrl+N).

Replace

Supposing we wanted to change all the Rem statements in a program with the equivalent "`" characters. This could be accomplished with the *Replace* command.

In order to use this option, it's necessary to define the replacement string. So the first time you call up replace, you will always be asked to enter this string from the keyboard.

Press Cntrl+R, type in ' (apostophe) at the prompt and hit the return key to enter it into the computer. You now set the search string with the *Find* option like so:

- Press Cntrl+F to select the Find option
- Type *Rem* into the information line
- The cursor will then be moved straight to the next Rem statement in your program listing.

To change this to the replacement string and jump to the next occurrence, select *Replace* (Cntrl+R) once again. Alternatively, if the Rem is in the middle of the line, you'll need to skip it, because AMOS only allows you to substitute a quote for this command at the start of a line. You can avoid this problem and jump directly to the next item in your program using *Find Next*.

Cut and paste

The AMOS Block commands allow you to cut out parts of your programs and save them in memory for future use. Once you've created a block, you can copy it anywhere you like in the current listing.

Here's an example of this feature in action. Let's take the previous ALERT program, and cut out a single procedure. Place the mouse pointer over the first line of the INVERT procedure, and depress the right mouse button. We can now enter this procedure into a block using the mouse. Hold down the right mouse key, and drag the point towards the bottom of the display. As you move the mouse, the selected area will be highlighted in reverse.

We can now grab this area into memory using *Cut*. When you press Cntrl+C from the keyboard, the procedure will be removed from the listing and stored into memory. It's now possible to *Paste* this block anywhere in your program. For the purposes of our example, move the text cursor down to the bottom of the listing, and call the Paste option with Cntrl+P. The INVERT procedure will now be copied to the current cursor position.

Multiple programs and accessories

Multiple programs

Although AMOS only allows you to edit a single program at a time, there's no limit to the number of programs which can be installed into memory, other than the amount of available storage space. Once you've installed a program in this way, you can execute it straight from Editor window with the *Run Other* option.

On an expanded system, you'll easily be able to store two or three full sized programs in memory without problems. But even if you're only using a standard A500 with 512k of memory, you'll still find a real use for this function.

Supposing, for instance, you encounter a problem in one of your programs. AMOS will let you effortlessly swap your existing program into memory so that you can freely experiment with the various possibilities until you find a solution. After you've finished, you can now grab your new routine into memory with the cut option, and flick back into your original program by pressing just two keys! The new routine can then be pasted into position, and you can continue with your program as before. This ability to stop everything and try out your ideas immediately, is incredibly valuable in practice.

Another possibility is to permanently keep all he most commonly needed utilities such as the sprite definer or the map editor in memory. You can now access these utilities instantaneously, whenever you need them.

In fact, AMOS includes a special ACCESSORY system which makes this even easier. The utility programs can be given total access to all the memory banks in your main programs. So the sprite definer can grab the images straight from your current program, and modify them directly. This technique speeds up the overall development process by an amazing degree!

Let's have a quick demonstration of these facilities. Enter the following small program into the editor:

```
Print "This is program One"
Boom
```

We can now *push* this program into memory using the push command. This is called up by pressing Amiga+P from keyboard. You will then be asked to enter the name of your program from the information line. Type a name like "Program1" at this point

The edit screen will be cleared completely.The new window is totally separate from your original program. As a demonstration, enter a second routine like so:

```
Print "This is program Two"
Shoot
```

This program can now be executed from the editor window using RUN (F1). But when your return you can immediately jump to the old one with the *Flick* option.

Try pressing Amiga+F from the keyboard. As before you'll be asked to enter a name for your program. Use a name like "program2" for this purpose. The editor will now jump straight to your original program as if by magic.

It's possible to repeat this process to jump back and forth between the two programs. Each program is entirely independent and can have it's list of own banks and program screens.

So far, we've only discussed how you can use two programs at a time. However, you can actually have as many program in memory as you like. These programs can be selected individually using the *Run Other* and *Edit Other* options from the Menu window. When you call these commands, a special "program" selector will be displayed on the screen.

The program selector is almost identical to the familiar AMOS file selector. The only difference is that is allows you to choose a program from memory rather than from the disc. You can select a program by simply highlighting it with the mouse cursor and clicking once on the left button.

Have a try at running and editing Program1 and Program two using this system. Once you get the hang of it, it's amazingly easy to use.. See the *Load Other* and *New Other* commands.

Accessories

In order to distinguish accessories from normal Basic programs, they're assigned a ".ACC" extension instead of the more usual ".AMOS". Accessories can be loaded into memory like any normal program using the *Load Other* command.

Load Other presents you with a normal fileselector which can be used to load an accessory program from the disc. After the accessory has been installed into memory you will be returned straight back to your current program . You can now run this accessory at any time using the Run Other option from the menu window. Simply move the mouse pointer over your required accessory and press the left button.

Alternatively, you can load all the accessories from the current disc using the *Accnew/ Load* feature. This option can be found on the System menu which is displayed when you hold down the right mouse button. *Accnew/Load* erases all existing accessories and loads a new set from the current disc.

For a demonstration, place the AMOS Program disc into your drive, and click on the AccNew/Load Button from the System menu.

The HELP accessory will be quickly loaded into memory. HELP is a special accessory because it is can be called up directly by pressing the HELP key on the keyboard. We've packed this program with all the information you'll need about the accessory programs supplied with AMOS Basic . All you need to do, is just follow the prompts which will be displayed on the screen.

Direct mode

The direct mode window can be entered from the editor by pressing the ESCape key at any time. As a default, the window is displayed in the lower half of the screen, with the program screen in the background.

If you run a program that changes the screen format, displays windows, animates sprites etc, then all this screen data will remain intact. So you can move the DIRECT window around or flip back to the editor to make program changes without destroying the current program screen. This DIRECT mode window is totally independent and is displayed on its own front level screen.

Whilst you're within direct mode you can type any line of AMOS Basic you wish. The only commands you cannot use are loops and branch instructions.You only have access to normal variables (as distinct from the local variables defined in a procedure).

Direct mode editor keys

ESCape	Jump to the editor window.
Return	Executes the current line of commands.
DELete	Delete character under cursor/
BACK SPACE	Delete character to the left of the cursor.
Left arrow	Move cursor left
Right arrow	Cursor right
Shift+left	Skip a word to the left
Shift+right	Skip a word to the right
Shift DELete	Deletes entire line.
Shift BACK	Ditto
Help	Displays the function key definitions to the direct window.
F1 to F10:	These keys remember the last 10 lines you've entered from direct mode. F1 displays the latest one entered, F2 the second to last etc. The memory area used by this system is always cleared when you return to the editor window or run one of your programs.

The menu window

Here's a detailed explanation of all the options which are available from the main menu window.

Default menu

This gives you various commands that allow you to operate the editor, plus give you access to the block and search menus.

Run (F1)
Runs the current program from memory.

Test (F2)
Checks the syntax of the entire program and places the cursor at the first error.

Indent (F3)
Takes the current program and neatly indents the listing for you.

Blocks Menu (Cntrl or F4)
Displays the Blocks menu in the selection window. These options can now be called either with the mouse or from the keyboard by pressing the appropriate function key. You can return to the main menu by simply clicking on the right mouse button, taking the pointer out of the function key area or by hitting a key.

Search Menu (Alt or F5)
Brings up the search menu. This allows you to search through your program for specific keywords and change them if required. You can also adjust the size of the text buffer or alter the current tab from this menu.

Run Other (F6)
Runs a program or accessory held in the Amiga's memory.

Edit Other (F7)
Edits a program which has been previously installed into memory using the *Load Other* or *Accnew/Load* commands. If you haven't saved your existing program, you will be prompted for it's name.

Your current program will now be pushed into memory, and you will be able to choose another program to edit using the program selector. To select this program, simply move the mouse over its name and press on the left button.

Note that the size of the editor buffer is stored into memory along with your program along with your program. This buffer area will reset to it's original size the next time you edit your program. If you attempt to edit a program which has not been saved in this way, such as an accessory, the memory buffer will be automatically increased if required. If you run out of memory the option will be aborted with an "out of memory" error.

Overwrite (F8)
Toggles between two separate editing modes.

Insert mode: (Default), inserts a space in your existing text to contain every character you type from the keyboard. If, for instance, you were editing a line like:

Rem TESTING INSRT MODE

Since the cursor is underneath the R, typing an
"E" would change the line to:

Rem TESTING INSERT MODE

Overwrite mode: Overwrite mode completely replaces the character under the cursor with your new keypress. Taking the previous example, typing "E" would produce the line:

Rem TESTING INSET MODE

After you've changed the mode, the menu item will be set to **Overwrite**. Selecting this option will return you back to the normal editing system.The current typing mode is indicated by a letter at the far left of the Information line. I=INSERT and O=OVERWRITE. Note that if you make a mistake while in insert mode, you can usually reverse the alterations on the current line by pressing Cntrl+U.

Fold/Unfold (F9)

Takes a procedure definition and folds it away inside your program listing. Once you've folded a procedure, only the first line in your definition will be displayed in your listings.

To fold a procedure, move the cursor anywhere inside the definition and select the *Fold* option. Your procedure will be tested for syntax errors, and will be hidden inside your program listing. It's vital to realise that absolutely **nothing** has happened to the actual program lines in the procedure. The only change has been in the way these lines are displayed in your listings.

Normally, it's possible to re-open a folded procedure by repeating the process. Place the cursor over a folded procedure and click on *Fold/Unfold.* (Hit F9 for a shortcut.) If you feel the need for extra security you can also call up a special LOCK accessory from the AMOS program disc. This will ask for a code word, and will lock your procedures so that they can't be subsequently examined from AMOS Basic. Simply fold your required procedures and load FOLD.ACC using the *Load Others* command. Full instructions are included with the utility.

The real beauty of this system is that it allows you to create whole libraries of your routines on the disc. These can be loaded into memory as a separate program (see LOAD OTHER). You can now cut out the routine you need and copy them directly into your main program. So once you've written a routine, you can place it into a procedure and reuse it again and again.

If you're intending to use this system, there are several points to consider.

- Whenever you fold or unfold a procedure a syntax check is made of the **entire** program. If an error occurs the operation will not be performed. So it's vital that you keep back-up copies of all your procedures in Unfolded format.

- Don't try to delete a folded procedure using the normal cursor keys. This will have no permanent effect. So define a block around your procedure and use CUT instead.

- Cut and Paste work fine with folded procedures. The complete procedure definition is entered into memory when you cut the Procedure statement. But you should take care to avoid the following errors.

"This array is not defined in your main program"

When you're copying a routine from one program to another, it's easy to forget about the SHARED or GLOBAL variables and arrays you've defined in your main program. If a procedure uses external variables, which have not ben defined, you'll get the above error. So check through your original listing for SHARED or GLOBAL statements.

"Label Defined Twice"

You've attempted to make TWO copies of a procedure in the same program! You've probably grabbed an extra procedure by mistake.

"Procedure not defined"

In AMOS Basic, it's perfectly acceptable to call procedures inside one another. This occasionally causes problems when you attempt to unpack one of these procedures, as the operation will only be performed if all the procedures it uses have been also been copied into your program. It's a good idea to list these routines at the start of a procedure, as this can avoid a **lot** of confusion!

Line insert (Cntrl+I or F10)
Inserts a line at the current cursor position.

The system menu

The SYSTEM menu contains a series of commands which allow you to load and save your programs from the disc. To select this menu simply press either the Shift key or hold down the right mouse button. Here's a full list of the available options.

Load (Shift+F1 or Amiga+L)
Loads an AMOS Basic file from the disc. This file is chosen using the standard AMOS file selector.

Save (Shift+F2 or Amiga+S)
Saves the current AMOS program. If you're saving a file for the first time, you will be asked to enter its name with the file selector. If there's another program on the disc with the same name, it will be automatically renamed with an extension of ".BAK". This provides a useful insurance against mistakes, as you can usually get back to the previous version of your program if you do something silly.

Save As (Shift+F3 or Shift+Amiga+S)
Saves the current program under another name. The new name is chosen with the help of the file selector. Note that If you save a program with the name AUTOEXEC.AMOS it

will be automatically loaded and executed on start-up. When you enter AMOS Basic, the main program screen will be displayed immediately.

Merge (Shift+F4)
Enters the chosen program at the current cursor position without erasing your original program first. This is used to incorporate routines taken from another program, such as the AMOS Map Definer.

Merge Ascii (Shift+F5)
Merges an Ascii version of an AMOS Basic program with the existing program in memory.

Ac.New/Load (Shift+F6)
Removes all current accessory routines from memory and enters a new set from the disc. All files with the extension ".ACC" will be automatically loaded by this command. If you're using an unexpanded A500, you should treat this command with a little caution as it's all too easy to run out of memory.

Load Others (Shift+F7)
Loads a single accessory program from the current disc. This can be accessed using the Run Others command from the main menu.

New Others (Shift+F8)
Erases one or more accessories from the Amiga's memory. When you call this command, you will be presented with a standard AMOS program selector. You can now click on either a single program to erase or the **all** button to delete all accessories.

New (Shift+F9)
Erases the program you are currently editing. If you haven't saved your program, you're given the option of saving it onto the disc before it is deleted.

Quit (Shift+F10)
Exits AMOS and returns control to the CLI. As with NEW you are given the option of saving your existing program before leaving AMOS.

The blocks menu

The Blocks menu provides you with the useful ability to move whole sections of your program from one place to another.

If required, these features can be accessed directly from the Main menu with the mouse. But you'll probably find it faster to use the alternative keyboard commands. Here's a complete list of the various options.

Block Start (Cntrl+B or Cntrl+F1)
Sets the starting point for the current block.

Block End (Cntrl+E or Cntrl+F6)
Defines the end of a block. Normally it's used straight after a Cntrl+B command. The region between your starting and ending points will now be displayed in inverse text.

Block Cut (Cntrl+C or Cntrl+F2)
Removes the selected block from its current position and loads it into memory. You can now copy this block anywhere in your program using the *Paste* command.

Block Paste (Cntrl+P Cntrl+F7)
Pastes the entire contents of a block at the current cursor position. This block must have been saved into memory using the *Cut* or *Store* commands.

Block Move (Cntrl+M or Cntrl+F3)
Moves the highlighted block straight to the current cursor position, erasing the original version completely.

Block Store (Cntrl+S or Cntrl+F8)
Copies the contents of a block into memory, without affecting the current program. This option provides you with a simple way of transferring lines from one program in memory to another.

Block Hide (Cntrl+H or Cntrl+F4)
Deselects the block you've highlighted using the Block Start and Block End commands.

Block Save (Cntrl+F9)
Saves the current block on the disc as a AMOS program. You can now reload it using the *Merge* or *Load* commands from the SYSTEM menu. Note any memory banks in your program are **not** saved along with your listing.

Save Ascii (Cntrl+F5)
Stores your selected block on the disc as a normal text file. This file can be loaded directly into any standard Wordprocessor. If you've access to a Modem, you can also copy these files on bulletin boards and communication networks such as MicroLink and Prestel.

Block Print (Cntrl+F10)
Outputs the selected block straight to the printer if it's connected.

There is also a special Select All command that can only be accessed via the keyboard. Pressing **Cntrl+A** will block select the whole of the current program. This is useful when you need to save out the entire program as an Ascii file.

The search menu

One of the best ways to learn about the AMOS system is to examine some of the example programs we've supplied on the AMOS Data disc. With the help of the AMOS Search menu, you can search the listings for examples of any AMOS instruction you like. This will give you valuable tips about how it can be used in the context of a real program.

You can also use the *Replace* command to change all the variable names in one of your Basic programs. So you can use short, simple names when you're entering your programs, and convert them into something more readable when you've finished.

The SEARCH menu can be called up straight from the menu window using the mouse. Alternatively, use one of the many keyboard short-cuts to call the required function directly.

Find **(Cntrl+F or Alt+F1)**
Enters a string of up to 32 characters from the keyboard and searches through your text until an exact match is found. The search precedes downwards from the current cursor position.

Find Next **(Cntrl+N or Alt+F2)**
Searches for the next occurrence of the string you specified using *Find.*

Find Top **(Alt+ F3)**
This is identical to *Find* except that it starts the search from the top of your program, rather than the cursor position.

Replace **(Alt+F4 or Cntrl+R)**
Activates *Replace* mode. The effect of this command varies depending on when it is used. There are two possibilities:

- **Before** a *Find* command

You will now be asked to enter the replacement string from the keyboard.

- **After** a *Find*

If the search operation was successful, the text and the current cursor position will be swapped with the replacement string. *Replace* will now jump to the next occurrence of the search string in your program. If you don't want to replace this item, you can skip directly to the next word with *Find Next.*

Replace All **(Alt+F5)**
Replaces **all** copies of a word in your program. The procedure is as follows:

- Confirm the command by hitting "Y" from the keyboard or clicking on the "Yes" box in the information line.

- Enter the string you wish to change.

- Input the string which the search string is to be replaced with. The search/replace will now proceed, starting from the **top** of your program.

Low<>Up **(Alt+F6)**
Changes the case sensitivity in your various search/replace commands. As a default all lowercase letters are distinguished from their equivalent CAPITALS. So "g" and "G" would be treated as different letters in your searches. This option forces AMOS to assume that the upper and lower case versions of your text are identical. In order to reflect the new operating mode, the display will be changed to **Low=Up.**

Open All **(Alt+F7)**
Opens all closed procedures in your program. A check is made of the syntax of the entire program before the procedures are unfolded. If an error is detected the operation will be

aborted.

If you encounter difficulties with this feature, see the *Fold* command for a detailed explanation of the possible problems and their solutions.

Close All (Alt+F8)

Closes all procedure definitions in your current program. Only the first line of your procedure definitions will be displayed in your listings. This makes them much shorter and removes much of the clutter from your listings. Like the previous *Open* command, the folding operation will only be performed if there are no errors in your current program.

Once you close your procedures you can edit them individually using the separate *Fold/Unfold* option.

Set Text B. (Alt+F9)

SET TEXT BUFFER. Changes the number of characters available to hold your listings. This can be used to increase the editors memory to allow you to enter particularly large programs into your Amiga. This can be extremely useful if you've added more memory to your Amiga. See also CLOSE EDITOR.

Set Tab (Cntrl+TAB or Alt+F10)

Sets the number of characters which the cursor will be moved when the user presses the TAB key.

Keyboard macros

AMOS Basic lets you create up to 20 keyboard macros at a time. These are accessed using a combination of the left or right Amiga key and a function key. Once you've defined a macro, it can be used anywhere in the AMOS system, just as if you'd entered your commands straight from the keyboard. The same macro can be called from the editor window, from direct mode, or even from inside one of your Basic programs!

The current key assignments are displayed in a special area above the direct mode window. While you are in direct mode, you can call this list on the screen with the HELP key. Similarly, pressing the left or right Amiga keys from the editor will display these definitions in the menu window.

As a default, all the macro assignments are loaded with a set of common Basic keywords. These can be changed using a simple option from the configuration accessory (CONFIG.ACC). It's also possible to assign these keys directly within one of your programs using the powerful KEY$ function.

=KEY$= *(Define a keyboard macro)*

```
KEY$(n)=command$
command$=KEY$(n)
```

KEY$ assigns the contents of *command$* to function key number *n*. *n* is an identification number of your function key from one to twenty.

Keys from one to ten are accessed by pressing the function key in conjunction with the left Amiga button. Similarly, numbers from eleven onwards are called with a right Amiga Fn combination.

Note that it's essential to press both keys simultaneously, otherwise your macro will be misinterpreted as two separate key presses.

command$ can be any string of text you wish, up to a maximum of 20 characters. There are two special characters which are directly interpreted by this function:

‘ (Alt+Quote)	Generates a Return code.
‘ (Single Quote)	Encloses a comment. This is only displayed in your key lists. It's totally ignored by the macro routine. Examples:

```
? Key$(1)
Key$(2)="Default‘"
Alt+F2

Key$(3)="‘Comment‘ Print"
```

In practice, this macro system can prove incredibly useful. Not only can you speed up the process of entering your Basic programs, but you can also define a list of standard inputs for your Basic programs. These would be extremely effective in an adventure game, as can be seen from the program **EXAMPLE 3.2** in the MANUAL folder.

If you wish to generate a keypress which has no Ascii equivalent such as up arrow, you can optionally include a scancode in these macros. This is achieved using the SCAN$ function.

=SCAN$ *(Return a scan code for use with KEY$)*

x$=Scan$(n [,m])

n is the scancode of a key to be used in one of your macro definitions. *m* is an optional mask which sets the special keys such as Cntrl or Alt, in the following format:

Bit	Key Tested	Notes
0	Left SHIFT key	
1	Right SHIFT key	
2	Caps Lock	Either ON or OFF
3	Control (Cntrl)	
4	Left Alt	
5	Right Alt	
6	Left Amiga	This is the Commodore key on some keyboards
7	Right Amiga	

If a bit is set to a one, then the associated button is "depressed " in your macro.
Examples:

```
KEY$(4)="Whee!"+Scan$($4C)
KEY$(5)="Page Up!"+scan$($4c,%00010000)
```

Conserving memory

If you're using an unexpanded A500, memory can occasionally get rather tight. We've therefore provided you with two powerful instructions which allow you to maximize the memory which is available for your programs.

Before discussing these functions, it's worth noting that if you have an external 3.5 inch drive, you can save around 30k by deactivating it before loading AMOS Basic.

Since AMOS only accesses the disc for a few small library files, you may find that the second drive is rarely if ever used. So it's sensible to take the memory and run!

Warning! Never turn off the drive while the Amiga is switched on. This will have absolutely no effect as the memory is allocated to the drive as part of your start-up sequence.

CLOSE WORKBENCH *(Closes the workbench)*

CLOSE WORKBENCH

Closes the workbench screen saving around 40K of memory for your programs! Example:

```
Print Chip Free,Fast Free
Close Workbench
Print Chip Free,Fast Free
```

CLOSE WORKBENCH can be executed either from direct mode, or inside one of your Basic programs. A typical program line might be:

```
If Fast Free=0 Then Close Workbench
```

This would check for a memory expansion and close the Workbench if extra memory was not available.

CLOSE EDITOR *(Close editor window)*

CLOSE EDITOR

Closes the Editor window while your program is running, saving you more than 28k of lovely memory. Furthermore, there's absolutely NO effect on your program listings!

If there's not enough memory to reopen the window after your program has finished, AMOS will simply erase your current display and revert back to the standard DEFAULT screen. You'll now be able to effortlessly jump back to the Editor with the Escape key as normal. What a terrific little instruction!

Inside accessories

We'll now explore the general techniques required to write your own accessory programs. These are really just a specialised form of the multiple programs we discussed a little earlier. As you would expect, they can incorporate all the standard Basic instructions.

Accessories are displayed directly over your current program screen and the music, sprite, or bob animations are automatically removed from the screen.

Your accessory should therefore check the dimensions and type of this screen using the SCREEN HEIGHT, SCREEN WIDTH and SCREEN COLOUR commands during its initialisation phase. If the current screen isn't acceptable, you may be forced to open a new screen for the accessory window or to erase the existing screens altogether with a DEFAULT instruction.

Any memory banks used by your accessory are totally independent of the main program. If it's necessary to change the banks from the current program, you can call a special BGRAB command.

BGRAB *(Grabs the banks used by the current program)*

BGRAB b

BGAB "borrows" a bank from the current program and copies it into the same bank in your accessory. If this accessory bank already exists, it will be totally erased. When the accessory returns to the editor, the bank you have grabbed will be automatically returned to your main program along with any changes. *b* is the number of a bank from 1 to 16.

Note that this instruction can only be used inside an accessory. If you try to include it in a normal program, you'll get an appropriate error message.

PRUN *(Run a program from memory)*

PRUN "name"

Executes a Basic program which has been previously installed in the Amiga's memory. This command can be used either from direct mode, or within a program! In effect, PRUN is very similar to a standard procedure call, except that any bobs, sprites or music will be totally suspended.

Note that it's impossible to call the same program twice in the same session. After you've called it once, any further attempts will be ignored completely.

When the program returns to your accessory you will need to restore your screen to its original state. This will avoid the danger of your accessory screens being corrupted by the new routine. See **EXAMPLE 3.3** in the MANUAL folder.

=PRG FIRST$ *(Read the first program loaded into memory)*

p$=PRG FIRST$

This returns the name of the first Basic program installed in the Amiga's memory. It's used in conjunction with the PRG NEXT$ command to create a full list of all the currently available programs.

=PRG NEXT$ *(Returns the next program installed in memory)*

p$=PRG NEXT$

PRG NEXT$ is used after a PRG FIRST$ command to page through all the programs installed in the Amiga's memory. When the end of the list is reached, a value of "" will be returned by this function. Here's an example:

```
N$=Prg First$
While N$<>“”
   Print “Program” “;N$
   N$=Prg Next$
Wend
```

=PSEL$ *(Call program selector)*

n$=PSEL$(“filter “[default$,title1$,title2$]

PSELS$ calls up a program selector which is identical to the one used by the Run Other, Edit Other, Load Others , and New Others commands. This can be used to select a program in the usual way. The name of this program will be returned in n$. If the user has aborted from the selector, n$ will be set to an empty string “”.

“filter” sets the type of programs which will be listed by this instruction. Typical values are:

“*.ACC”	List all the accessories in memory.
“*.AMOS”	Only displays the AMOS programs which have been installed.
“.*”*	List all programs currently in memory.

For further details of the system see the DIR command.

default$	holds the name of a program which will be used as a default.
title1$,title$	Contains up to two lines of text which will be displayed at the top of the selector.

See **EXAMPLE 3.4** in the MANUAL folder for a demonstration of this instruction.

The HELP accessory

Whenever the HELP key is pressed from the Editor window, AMOS automatically executes an accessory with the name HELP.ACC if it's available. Unlike normal accessories, this is displayed directly over the editor window. Special access is provided to the current word you are editing. The address of this word is placed in an address register and can be read using the AREG function.

The editor control keys

Finally, here's a full list of the various control keys and their effects.

Special keys

ESCape	Takes you to direct mode

Editing keys

Backspace	Deletes the character to the immediate left of the cursor.
DELete	Deletes the character directly underneath the cursor.
RETURN	Tokenises the current line. If you move onto a line and press RETURN it will split the line (this only takes effect if you haven't changed anything).
Shift+Back or Cntrl+Y	Deletes current line and then pulls the rest of the text up from below.
Cntrl+U	Undo. Return the last line when in Overwrite mode.
Cntrl+Q	Erase the rest of the characters in the line starting from the present cursor position.
Cntrl+I	Insert a line at the current position.

The cursor arrows

Left	Cursor one space to the left.
Right	Cursor one space to the right.
Up	Moves the cursor up by one line. There's no effect if you are at the top line of your program.
Down	Move the cursor down by a line.
Shift+left	Place the cursor over the previous word.
Shift+right	Position the cursor over the next word.
Shift+up	Move the cursor to the top line of the present page.
Shift+down	Transport the cursor to the bottom line of the current page.
Cntrl+up	Display the previous page of text.
Cntrl+down	Display the next page of your program.
Shift+Cntrl+up	Move to start of text.
Shift+Cntrl+down	Jump to end of text.
Amiga+up	Scrolls text up without moving the cursor.
Amiga+down	Scrolls text down under the cursor.
Amiga+left	Scroll program to the left. The cursor stays fixed on the current line.
Amiga+right	Moves text to the right.

Program control

Amiga+S	Saves your program under a new name.
Amiga+shift+S	Saves program under current name.
Amiga+L	Loads a program.
Amiga+P	Pushes the current program into memory and creates a new program.
Amiga+F	Flips between two programs stored in memory.
Amiga+T	Displays next program in memory. Repeating this option will allow you to display all the programs currently in memory.

Cut and Paste

Cntrl+B	Set the beginning of a block.
Cntrl+E	Set end point of a block.
Cntrl+C	Cut block. Loads block into memory and erases it from its current position. (This combination also works in DIRECT MODE, where it HALTS to the current program.)
Cntrl+M	Block move.
Cntrl+S	Saves the block in memory without erasing it first.
Cntrl+P	Paste block at current cursor position.
Cntrl+H	Hide block. The highlighting will be removed from the chosen block.

Marks

Cntrl+shift+n	Defines a marker at the present cursor position. *n* must be a digit from the numeric keypad in the range 0 to 9.
Cntrl+n	Go to mark. Jumps to a previously set mark. *n* must be from the numeric keypad.

Search/Replace

Alt+up	Searches backwards through your program to the next line which contains label or procedure definition. If AMOS has reached the end of your procedures, the cursor will remain at its current position.
Alt+down	Searches down through your program to find the next label or procedure definition.
Cntrl+F	Find. Asks you to enter some text to be searched for in your program. Then jumps to the first copy it can find, starting from the current cursor position.
Cntrl+N	Find Next. Use this after a find to jump to the next occurrence of your string.
Cntrl+R	Replace. If you use this prior to *Find*, you will be prompted for a replacement string. After you've started a search with *Find* however, Cntrl+R will replace your word with the new text, and jump to the next occurrence of the search string.

Tabs

Tab	Move the entire line at the current cursor to the next Tab stop.
Shift+Tab	Move the line to the previous Tab.
Cntrl+Tab	Sets the Tab value.

4: Basic principles

This chapter discusses the ground rules used to construct AMOS Basic programs and shows you how to improve your programming style with the help of AMOS Basic procedures.

Variables

Variables are the names used to refer to storage locations inside a computer. These locations hold the results of the calculations performed in one of your programs.

The choice of variable names is entirely up to you, and can include any string of letters or numbers. There are only a couple of restrictions. All variable names **must** begin with a letter and cannot commence with an existing AMOS Basic instruction. However it is perfectly permissible to use these keywords inside a name. So variables such as VPRINT or SCORE are fine.

Variable names must be continuous, and may not contain embedded spaces. If a space is required, it's possible to substitute a "_" character (Shift minus) instead.

The following are examples of legal names:

```
AWHILE$, HIGH_SCORE, TEST_FLAG, HEIGHT#
```

The maximum length of these variable names is 255 characters, for example:

```
A_VERY_LONG_NAME=10 : Rem This is ok
```

Here are some examples of illegal names. The illegal bits are underlined to make things clearer.

```
WHILE$, 5C, MODERN#, TOAD
```

Types of variables

AMOS Basic allows you to use three different types of variables in your programs.

Integers

Unlike most other Basics, AMOS initially assumes that all variables are integers. Integers are whole numbers such as 1, 3, or 8, and are ideal for holding the values used in your games.

Since integer arithmetic is much faster than the normal floating point operations, using integers in your programs can lead to dramatic improvements in speed. Each integer is stored in four bytes and can range from -147,483,648 to +147,483,648. Examples of integer variables:

```
A, NUMBER, SCORE, LIVES
```

Real numbers

In AMOS Basic these variables are always followed by a hash (#) character. Real numbers can hold fractional values such as 3.1 or 1.5. They correspond directly to the standard variables used in most other versions of Basic. Each real variable is stored in four bytes and can range between 1E-14 and 1E-15. All values are accurate to a precision of seven decimal digits. Examples:

```
P#, NUMBER#, TEST#
```

String variables

String variables contain text rather than numbers. They are distinguished from normal variables by the $ character at the end. The length of your text can be anything from 0 to 65,500 characters. Examples of string variables:

```
NAME$, PATH$, ALIEN$
```

Giving a variable a value

Assigning a value to a variable is easy. Simply choose an appropriate name and assign it to a value using the "=" statement.

```
VAR=10
```

This loads the variable VAR with a value of 10. Depending on the type of your variable it can contain either a number or a list of characters. To assign a *string* to a variable, you enclose it with a pair of double quotes like so:

```
A$="Hello"
```

Notice the $ sign after the name. This tells AMOS that the variable will contain characters rather than a number.

Arrays

Any list of variables can be combined together in the form of an *array.* Arrays are created using the DIM instruction.

DIM *(Dimension an array)*

DIM var(x,y,z,...)

DIM defines a table of variables in your AMOS Basic program. These tables may have as many dimensions as you want, but each dimension is limited to a maximum of 65,000 elements. Example:

```
Dim A$(10),B(10,10),C#(10,10,10)
```

In order to access an element in the array you simply type the array name followed by the index numbers. These numbers are separated by commas and are enclosed between round brackets (). Note that the element numbers of these arrays always start from zero. Example:

```
Dim ARRAY(10)
ARRAY(0)=10:ARRAY(1)=15
Print ARRAY(1);ARRAY(0)
15 10
```

Constants

Constants are simply numbers or strings which are assigned to a variable or used in one of your calculations. They are called constants because they don't change during the course of your program. The following values are all constants:

1, 42, 3.141, "Hello"

As a default, all numeric constants are treated as integers. Any floating point assignments to an integer variable are automatically converted to a whole number before use. Examples:

```
A=3.141:Print A
3
Print 19/2
9
```

Constants can also be input using binary or hexadecimal notation. Binary numbers are signified by preceding them with a % character, and hexadecimal numbers are denoted by a $ sign. Here's an example of the various different ways the number 255 could be expressed.

Decimal:	**255**
Hexadecimal:	**$FF**
Binary:	**%11111111**

Note that any numbers you type into AMOS Basic are automatically converted into a special internal format. When you list your program these numbers are expanded back into their original form. Since AMOS Basic prints all numbers in a standard way, this will often lead to minor discrepancies between the number you entered and the number which is displayed in your listing. However the value of the number will remain exactly the same.

Floating point constants are distinguished from integers by a decimal point. If this point is not used, the number will always be assumed to be an integer, even if this number occurs inside a floating point expression. Take the following example:

```
For X=1 To 10000
   A#=A#+2
Next X
```

Every time the expression in this program is evaluated, the "2" will be laboriously converted into a real number. So this routine will be inherently slower than the equivalent program below.

```
For X=1 To 10000
  A#=A#+2.0
Next X
```

This program executes over 25% faster than the original one because the constant is now stored directly in floating point format. You should therefore always remember to place a decimal point after a floating point constant even if it is a whole number. Incidentally, if you mix floating point numbers and integers, the result will always be returned as a real number. Example:

```
Print 19.0/2
9.5
Print 3.141+10
13.141
```

Arithmetic operations

The following arithmetic operations can be used in a numeric expression.

^	Power
/ and *	Divide and multiply
MOD	Modulo operator (remainder of a division)
+ and -	Plus and minus
AND	Logical AND
OR	Logical OR
XOR	Logical XOR

We've listed these operations in descending order of their priority. This priority refers to the sequence in which the various sections of an arithmetic expression are evaluated. Operations with the highest priority are always calculated first. Here is an example of how this works in practice:

```
Print 10+2*5-8/4+5^2
```

This evaluates in the following order:

5^2 (Equal to 5*5) = 25
2*5 = 10
8/4 = 2
10+10 = 20
20-2 = 18
18+25 = 43

If you wanted this to evaluate differently, you would simply enclose the parts of the expression you wished to execute first in round brackets:

```
Print (10+2)*(5-8/4+5)^2
```

This gives the result 12*(8^2) or 12*64 or 768. As you can see, the addition of just two pairs of brackets has changed the sense of the expression completely.

While on the subject of arithmetic, it's worth mentioning three simple instructions which can speed up your programs considerably.

INC *(Add 1 to an integer variable)*

INC var

INC adds 1 to an integer variable using a single 68000 instruction. It is logically equivalent to the expression var=var+1, but is much faster. Example:

```
A=10:Inc A:Print A
11
```

DEC *(Subtract 1 from an integer variable)*

DEC var

This instruction subtracts 1 from the integer variable *var*. Example:

```
A=2
Dec A
Print A
1
```

ADD *(Fast integer addition)*

ADD v,exp [, base TO top]

The standard form of this instruction immediately adds the result of the expression *exp* to the integer variable *v*. It's equivalent to the line: V=V+EXP

The only significant difference between the two statements is that ADD performs around 40% faster. Note that the variable *v* must be an integer. Example:

```
Timer=0
For X=1 To 1000
  Add T,X
Next X
Print T,Timer
500500 7
```

The second version of ADD is a little more complicated. It is effectively identical to the following code:

```
V=V+A
```

```
If V<Base Then V=TOP
If V>Top Then V=BASE
```

Like the first version of ADD this command is considerably faster than the separate instructions. Here's an example:

```
Dim A(10)
For X=0 To 10:A(X)=X:Next X
V=0
Repeat
  Add V,1,1 To 10
Print A(V)
Until V=100:rem This is an infinite loop as V is always less than 10!
```

As you can see, ADD is ideal for handling circular or repetitive loops in your games.

String operations

Like most versions of Basic, AMOS will happily allow you to add two strings together.

```
A$="AMOS"+" Basic"
Print A$
AMOS Basic
```

But AMOS also lets you perform subtraction as well. This operation works by removing all occurrences of the second string from the first. Examples:

```
Print "AMOS BASIC"-"S"
AMO BAIC
Print "AMOS BASIC"–"AMOS"
 BASIC
Print " A String of Characters"-" "
AStringofCharacters
```

Comparisons between two strings are performed on a character by character basis using the Ascii values of the appropriate letters. Examples:

```
"AA" < "BB"
"Filename"="Filename"
"X&" > "X#"
"HELLO" < "hello"
```

Type in the following program:

```
Input "Enter your First name";C$
Input "Enter your Surname ";S$
If C$>S$ Then Print S$;"   ";C$ Else Print C$;"   ";S$
```

Parameters

The values you enter into an AMOS Basic instruction are known as *parameters.* i.e

```
Inc N
Add A,10
Ink 1,2,3
```

The parameters in the above instructions are N, A,10,1,2 and 3 respectively. Occasionally, some of the parameters of a command can be omitted from an instruction. In this case, any unused values will automatically be assigned a number by default. Take the following example:

```
Ink 5,,
```

This changes the ink colour without affecting either the paper or outline colours. Notice the commas in their normal positions, even though the values themselves have been omitted. AMOS uses these commas to work out which order the parameters are to be entered into the instruction. This allows you to input a value in the middle of a command like so:

```
Ink ,3,
```

Ink now sets the paper colour leaving the ink and outline colours untouched.

The same principle can also be applied to many other AMOS Basic instructions. Providing you remember to keep the commas in their original positions, you can use this technique to avoid a great deal of unnecessary typing in your programs. If a parameter is essential you'll be presented with an *illegal function call.* So it's well worth experimenting with the various combinations.

Line numbers and labels

Earlier versions of Basic expected each program line to begin with a number. This *line number* served as a target for the GOTO or GOSUB instructions. It was also used by the Basic editor. While there's nothing wrong with this approach (it was even used in STOS Basic), it's not really necessary with AMOS. In AMOS Basic all line numbers are completely optional; they are only provided for compatibility purposes with STOS Basic.

You may be wondering how you can use GOTO or GOSUB without line numbers. Well, you can replace them using *labels.*

Labels

Labels are just a convenient way of marking a point in your AMOS Basic programs. They consist of a string of characters formed using the same rules as AMOS variables. Labels should always be placed at the start of a line, and must be followed immediately by a ":" (colon) character. There should be no spaces between the label and the colon. Otherwise the label will be treated as a procedure and you'll get an *Undefined procedure* error. Here's a simple example:

```
TESTLABEL: Rem This is a label
Print "Hi there"
Goto TESTLABEL
```

This program repeatedly prints the words "Hi there" on the screen. It can be aborted by pressing Control+C.

Labels are much easier to read than line numbers. You are therefore advised to use them extensively in your AMOS Basic programs.

Procedures

If you've ever attempted to write a really large Basic program, you'll appreciate how easy it can be to get completely lost halfway through. Nowadays most professional programmers split their programs into small modules known as procedures.

Procedures allow you to concentrate your efforts on just one problem at a time without the distractions provided by the rest of your program. Once you've written your procedures you can then quickly combine them in your finished program.

Programs which use procedures are easy to write, easy to change and easy to debug. AMOS Basic procedures are totally independent program modules which can have their own program lines, variables, and even data statements. So there's absolutely no excuse for not making full use of them in your AMOS Basic programs.

PROCEDURE *(Create an AMOS Basic procedure)*

```
Procedure NAME[parameter list]
:        :
End Proc[Expression]
```

This defines an AMOS Basic procedure called *NAME. NAME* is a string of characters which identify the procedure. It is constructed in exactly the same way as a normal Basic variable. Note that it's perfectly acceptable to use identical names for procedures, variables and labels. AMOS will automatically work out which object you are referring to from the context of the line.

Procedures are similar to the GOSUB commands found in earlier versions of Basic. Here's an example of a simple AMOS procedure.

```
Procedure ANSWER
  Print "Forty-Two!"
End Proc
```

See how the procedure has been terminated with an *END PROC* statement. You should also note that the Procedure and the End Proc directives are both placed on their own separate lines. This is compulsory.

If you type the previous procedure into AMOS Basic as it stands, and attempt to run it, nothing will happen. That's because you haven't actually called the new procedure from your Basic program. This can be achieved by simply entering its name at the appropriate point in the program. As an example, enter the following line at the start of the program and run it to see the result of the procedure.

```
ANSWER
```

Important! When you are using several procedures on the same line, it's advisable to add an extra space at the end of each statement. This will avoid the risk of the procedure being confused with a label. For example:

```
TEST : TEST : TEST:Rem performs the test procedure three times
TEST:TEST:TEST: Rem defines the label TEST and executes TEST just twice
```

Alternatively, you can preclude your Procedure calls with a Proc statement like so:

```
Proc ANSWER
```

Example:

```
Rem Shows you for sure that it is a Procedure being called,
Rem not just a command.
Proc ANSWER
Rem The same can be achieved without the Proc
ANSWER
Procedure ANSWER
  Print "Forty-Two!"
End Proc
```

If you run this program again, the procedure will be entered, and the answer will be printed out on the screen. Although the procedure definition is positioned at the end of the program, it's possible to place it absolutely anywhere. Whenever AMOS encounters a Procedure statement, it installs the procedure and immediately jumps to the final End Proc. This means there is no danger of accidentally executing your procedure by mistake. Once you've created a procedure, and tested it to your satisfaction, you can suppress it in your listings using the *fold* option from the main menu.

These *folding* procedures reduce the apparent complexity of your listings and allow you to debug large programs without the distractions of unimportant details. You can restore your procedure listings to the screen at any time by selecting the *unfold* menu option.

Local and global variables

All the variables you define inside your procedures are independent of any other variables used in your program. These variables are said to be *local* to your particular procedure. Here's an example which illustrates this:

```
A=1000:B=42
TEST
Print A,B
Procedure TEST
  Print A,B
End Proc
```

It should be apparent that the names A and B refer to completely different variables depending on whether they are used inside or outside the procedure TEST. The variables which occur outside a procedure are *global* and cannot be accessed from within it. Let's take another example:

```
Dim A(100)
For V=1 to 100: A(V)=V:Next V
TEST_FLAG=1
APRINT
End
Procedure APRINT
  If TEST_FLAG=1
     For P=1 To 100
        Print A(P)
     Next P
  Endif
Endproc
```

This program may look pretty harmless but it contains two fatal errors.

Firstly, the value of TEST_FLAG inside the procedure will always have a value of zero. So the loop between the IF and the ENDIF will never be performed. That's because the version of TEST_FLAG within the procedure is completely separate from the copy defined in the main program. Like all variables, it's automatically assigned to zero the first time it's used.

Furthermore, the program won't even run! Since the global array A() has been defined outside APRINT, AMOS Basic will immediately report an *array not dimensioned* error at the line:

```
Print A(P)
```

This type of error is extremely easy to make. So it's vital that you treat procedures as separate programs with their own independent set of variables and instructions. Don't fall into the trap of using the same variable names inside and outside a procedure. Otherwise you could be hoodwinked into believing they are the same variables, which could lead to inexplicable errors in your programs.

Fortunately, there are a couple of extensions to this system which make it easy for you to transfer information between a procedure and your main program. Once you're familiar with these commands you'll have few problems in using procedures successfully in your programs.

Parameters and procedures

One possibility is to include a list of "parameter definitions" in your procedure. This creates a group of local variables which can be loaded directly from the main program. Here's an example:

```
Procedure HELLO[NAME$]
  Print "Hello ";NAME$
End Proc
```

The value to be loaded into NAME$ is entered between square brackets as part of the procedure call. So the HELLO procedure could be performed in the following ways:

```
Rem Loads N$ into NAME$ and enters procedure
Input "What's your name";n$
HELLO[n$]
Rem Load the literal string "Stephen" into NAME$ and call HELLO
HELLO["Stephen"]
```

As you can see, the parameter system is general purpose and works equally well with either variables or constants. Only the type of the variables are significant.

This process can be used to transfer integer, real or string variables. However you cannot pass entire arrays with this function. If you want to enter several parameters you should separate your variables using commas. For example:

```
Procedure POWER[A,B]
Procedure MERGE[A$,B$,C$]
```

These procedures might by called using lines like:

```
POWER[10,3]
MERGE["One","Two","Three"]
```

Shared variables

Another way of passing data between a procedure and the main program is to use the SHARED instruction.

SHARED *(Define a list of global variables)*

SHARED variable list

SHARED is placed inside a procedure definition and takes a list of AMOS Basic variables separated by commas. These variables are now treated as *global* variables, and can be accessed directly from the main program. Any arrays which you declare in this way should of course have been previously dimensioned in your main program. Example:

```
A=1000:B=42
TEST
Print A,B
Procedure Test
  Shared A,B
  A=A+B:B=B+10
End Proc
```

TEST can now read and write information to the global variables A and B. If you want to share an array you should define it like so:

```
Shared A(),B#(),C$():rem Shares arrays A,B# and C$
```

GLOBAL *(Declare a list of global variables from the main program)*

GLOBAL variable list

When you're writing a large program, it's commonplace for a number of procedures to share the same set of global variables. This provides a simple method of transferring large amounts of information between your various procedures. In order to simplify this process, we've included a single command which can be used directly in your main program. GLOBAL defines a list variables which can be accessed anywhere inside your Basic program, without the need for an explicit SHARED statement in your procedure definitions. Example:

```
A=1000 : B=42
Global A,B
TEST1
Print A,B
TEST2
Print A,B
Procedure TEST1
   A=A+B : B=B+10
End Proc
Procedure TEST2
   A=A*B : B=B+10
End Proc
```

Returning values from a procedure

If a procedure needs to return a value which is only local to itself, it must use the following command so that it can inform the calling PROCEDURE command where to find the local variable.

PARAM *(Return a parameter from a procedure)*

PARAM

The PARAM functions provide you with a simple way of returning a result from a procedure. They take the result of an optional expression in the END PROC statement, and return it in one of the variables PARAM,PARAM#, or PARAM$ depending on its type. Example:

```
MERGE_STRINGS[“Amos”,” “,”Basic"]
Print PARAM$
Procedure MERGE_STRINGS[A$,B$,C$]
  Print A$,B$,C$
End Proc[A$+B$+C$]
```

Note that END PROC may only return a single parameter in this way. The PARAM functions will always contain the result of the most recently executed procedure.Here's another example, this time showing the use of the PARAM# function.

```
CUBE[3.0]
Print Param#
Procedure CUBE[A#]
  C#=CUBE#*CUBE#*CUBE#
Endproc[C#]
```

Leaving a procedure

POP PROC *(Leave a procedure immediately)*

POP PROC

Normally, procedures will only return to the main program when the END PROC instruction is reached. Sometimes, however, you need to exit from a procedure in a hurry. In this case you can use the POP PROC function to exit immediately. Example:

```
Procedure TERMINATE
  ForBORING=1 to 100000
    IfBORING=10 Then Pop Proc
  NextBORING
  Print "This line is never executed"
End Proc
```

Local DATA statements

Any data statements defined inside one of your procedures are held completely separately from those in the main program. This means each procedure can have its own individual data areas. Example:

```
Read A$:Print A$,
EXAMPLE
Read B$:Print B$,
Procedure EXAMPLE
  Read X$,Y$
  Print X$,Y$
  Data "Basic", "is"
End Proc
Data "AMOS","amazing!"
```

Hints and tips

Here are a few guidelines which will help you make the most out of your AMOS Basic procedures:

- It's perfectly legal for a procedure to call itself, but this recursion is limited by the amount of space used to store the local variables. If your program runs out of memory you'll get an appropriate error.

- All local variables are automatically **discarded** after the procedure has finished executing.

```
Procedure ADD
  A=A+1:Print A
End Proc
```

No matter how many times you call this procedure, it will always print the same value (1).

- AMOS procedures are equivalent to subroutines created using the Amiga Basic SUB commands. The only significant difference is that you can't pass arrays as parameters. If you need to access an array from within a procedure, you should declare it as *shared* instead.

Memory banks

AMOS Basic includes a number of powerful facilities for manipulating sprites, bobs and music. The data required by these functions needs to be stored along with the Basic program. AMOS Basic uses a special set of 15 sections of memory for this purpose called *banks.*

Each bank is referred to by a unique number ranging from 1 to 15. Many of these banks can be used for all types of data, but some are dedicated solely to one sort of information such as sprite definitions. All sprite images are stored in bank 1. They can be loaded into memory using a line like:

```
Load "AMOS_DATA:Sprites/Octopus.abk"
```

There are two different forms of memory bank: *Permanent* and *temporary.* Permanent banks only need to be defined once, and are subsequently saved along with your program automatically. Temporary banks are much more volatile and are reinitialised every time a program is run. Furthermore, unlike permanent banks, temporary banks can be erased from memory using the CLEAR command.

Types of memory bank

AMOS Basic supports the following types of memory bank:

Class	Stores	Restrictions	Type
Sprites	Sprite or bob definitions	Only bank 1	Permanent
Icons	Holds icon definitions	Only bank 2	Permanent
Music	Contains sound track data	Only bank 3	Permanent
Amal	Used for AMAL data	Only bank 4	Permanent
Samples	The Sample data	Banks 1-15	Permanent
Menu	Stores MENU definition	Banks 1-15	Permanent
Chip Work	Temporary workspace	Banks 1-15	Temporary
Chip Data	Permanent workspace	Banks 1-15	Permanent
Fast Work	Temporary workspace	Banks 1-15	Temporary
Fast Data	Permanent workspace	Banks 1-15	Permanent

RESERVE *(Reserve a bank)*

RESERVE AS type, bank, length

The banks used by your sprites or bobs are allocated automatically by AMOS.The RESERVE command allows you to create any other banks which you might require. Each different type of bank has its own unique version of the RESERVE instruction.

RESERVE AS WORK bankno,length

Reserves *length* bytes for use as a temporary workspace. Whenever possible this memory area will be allocated using fast memory, so you shouldn't call this command in conjunction with instructions which need to access the Amiga's blitter chip.

RESERVE AS CHIP WORK bankno,length.

Allocates a workspace of size *length* using chip ram. You can check whether there's enough chip ram available with the CHIP FREE function.

RESERVE AS DATA bankno,length

Reserves a permanent bank of memory *length* bytes long. This data area will be allocated using fast memory if it's available.

RESERVE AS CHIP DATA bankno,length

Reserves *length* bytes of memory from chip ram. This bank will be automatically saved along with your AMOS programs.

bank may be any number between 1 and 15. Since banks 1 to 5 are normally reserved by the system, it's wisest to leave them alone. Note that the only limit to the length of a bank is the amount of available memory.

LISTBANK *(List the banks in use)*

LISTBANK lists the numbers of the banks currently reserved by a program, along with their location and size. The listing is produced in the following format:

```
Number   Type      Start        Length

1      - Sprites   S: $040F60   L:$00002F
2      - Work      S: $05F7A0   L:$014000
```

S: = The start address of the bank in hexadecimal.
L: = The length of the bank in hexadecimal.

Normally the length of a bank is returned in bytes, but in the case of sprites and icons the value represents the total number of images in the bank instead. The reason for this is that the storage of each image can be anywhere in the Amiga's memory, the bank is therefore not a continuous block of memory. So don't BSAVE a sprite bank, simply use SAVE "filename.ABK".

Deleting banks

During the course of a program you may need to clear some banks from memory so as to load in additional data. Sprites may need to change for a new part of a game or a special piece of music is required to be played. The ERASE command gives you quick control for data deletion.

ERASE *(Delete a bank)*

ERASE b

ERASE deletes the contents of a memory bank. The bank number *b* can range from 1 to 15. Note that any memory used by this bank is subsequently freed for use by your program.

Bank parameter functions

If you want to have direct access to the bank data using commands such as poke, doke and loke then use these commands to find a bank's address in memory and its size.

=START *(Get the start address of a bank)*

s=START(b)

This function returns the start address of bank number *b*. Once it's been reserved, the location of the bank will never subsequently change. So the result of this function will remain fixed for the lifetime of the bank. Example:

```
Reserve As Work 3,2000
Print Start(3)
```

=LENGTH *(Get the length of a bank)*

l=LENGTH(b)

The LENGTH function returns the length in bytes of bank number *b*. If the bank contains sprites then the number of sprites or icons will be returned instead. A value of zero indicates that bank *b* does not exist. Example:

```
Reserve as work 6,1000
Print Length(6)
Erase 6
Print Length(6)
```

Loading and saving banks

Some programs will require many banks of information, a good example is an adventure. This would need to load various graphics and sounds for the different locations within the games domain. An Amiga 500 would have great difficulty holding all this data at once and so it's best to simply load the data at the appropriate time of use.

LOAD *(Load one or more banks)*

LOAD "filename"[,n]

The effect of this command varies depending on the type of file you are loading. If the file holds several banks, then **all** current memory banks will be erased before the new banks are loaded from the disc. However if you're loading just a single bank, only this bank will be replaced. The optional destination point specifies the bank which is to be loaded with your data. If it's omitted, then the data will be loaded into the bank from which it was originally saved.

Sprite banks are treated slightly differently. In this case the parameter *n* toggles between two separate loading modes. If *n* is omitted or is assigned a value of zero, the current bank will be completely overwritten by the new sprites. Any other value for *n* forces the new sprites to be **appended** to this bank. This allows you to combine several sprite files into the same program. Example:

```
Load "AMOS_DATA:Sprites/Octopus.abk"
```

SAVE *(Save one or more banks onto the disc)*

SAVE"filename"[,n]

The SAVE command saves your memory banks onto the disc. There are two possible formats:

```
SAVE "filename.ABK"
```

This saves **all** the currently defined banks into a single file on your disc.

```
SAVE "filename.ABK",n
```

The expanded form just saves memory bank number *n*. One should also be sure to use the extension *ABK* at the end of the filename as this will ensure you can identify that the file contains one or more memory banks.

BSAVE *(Save an unformatted block of memory in binary format)*

BSAVE file$, start TO end

The memory stored between *start* and *end* is saved on the disc in *file$*. This data is saved with no special formatting. Example:

```
Bsave "Test",Start(7) TO Start(7)+Length(7):rem Saves a memory bank
```

The above example saves the data in memory bank 7 to disc. The difference between this file and a file saved as a normal bank is that SAVE writes out a special bank header that contains information concerning the bank. This header is not present with a BSAVED file so it cannot be loaded using LOAD.
Warning: The sprites and icon banks are not stored as one chunk of memory. Each object can reside anywhere in memory. Because AMOS uses this flexible system of data storage you simply can't save the memory bank using BSAVE.

BLOAD *(Load binary information into a specified address or bank)*

BLOAD file$, addr

The BLOAD command loads a file of binary data into memory. It does not alter the incoming information in any way. There are two forms of this function.

Bload file$, addr

File$ will be loaded from the disc into the address *addr.*

Bload file$, bank

File$ will be loaded into *bank*. This bank must have been previously reserved, otherwise an error will be generated. Also be sure not to load a file that is larger than the reserved bank, otherwise it will over run the bank and start corrupting other areas of memory.

Memory fragmentation

Sometimes, after a busy editing session, you may get an "Out of memory" error, even though the Information line implies that you have plenty of available memory.This is **not** a bug in AMOS Basic. It's just an unavoidable side effect of the Amiga's memory allocation system.

The Amiga's memory system is rather like a cake. Once you've cut a slice there's absolutely no way of replacing it in it's original position.Every time you reserve some memory,a single slice is cut from the memory cake.When you return this memory back to the system, it's placed on a special"stack" of all the currently unused slices. (Maybe its a *current* cake!)

The next time you reserve an area of memory, the Amiga will check each slice in turn for a region of the required size.If it finds a section which is larger than the one you requested,it will automatically cut it into two pieces, keeping any unused memory for itself. After a while, the memory will fragment into a large number of very small slices. So when you ask for some more, there's no way of allocating it from a single chunk, and you'll get an annoying "Out of memory" error.

The only reliable way of solving this problem, is to turn off your Amiga and reboot. This will restore the memory area to it's original, state, and you'll be able to allocate memory as and when you need it.

Note that the above difficulty only crops up during development. Providing you reserve all your memory at the start of your programs, you'll never have to worry about the problem in one of your actual programs.

Finding space for your variables

As a default, all variables are stored in a memory area of exactly 8k in length. Although this may seem incredibly meagre, it's easily capable of holding around 2 pages of normal text, or 2000 numbers. We've intentionally set it as small as possible so as to maximize the amount of space available for your screens and memory banks.

Don't Panic however! The size of this area can be increased directly from within your Basic programs using a simple SET BUFFER command. So the only physical limit to the size of your arrays and string variables will be the amount of memory you've installed in your computer.

SET BUFFER *(Set the size of the variable area)*

SET BUFFER n

Sets the size of the variable area in your current program to *n* kilobytes. This must be the FIRST instruction in your program (excluding Rems). Otherwise you'll get an appropriate error message. For an example of this feature see **EXAMPLE 4.1** in the MANUAL folder.

SET BUFFER should be used in your program whenever you get an *out of string space* error. Increase the value in 5k increments until the error disappears. If you run out of memory during this process, you'll probably need to reduce the requirements of your program in some way. See the CLOSE WORKBENCH and CLOSE EDITOR commands for more details.

=FREE *(Return the amount of free memory in the variable area)*

f=FREE

FREE returns the number of bytes which are currently available to hold your variables. This value can be increased as required using the previous SET BUFFER command.

Whenever FREE is called, the variable area is reorganized to provide the maximum space for your variables. This process is known as *garbage collection*, and is normally performed automatically.

Due to the power of AMOS Basic, the entire procedure is usually accomplished practically instantaneously. But If you're variable area is very large and you're using a lot of strings, the garbage collection routine might take several seconds to complete. Conceivably, this could lead to a unexpected delay in the execution of your programs. Since the garbage collection is totally essential, (just as in real life!) you may need to add an explicit call to the FREE command when it will cause the least amount of harm in your program.

5: String functions

AMOS Basic comes complete with a full range of string manipulation instructions. We've taken care to use the standard Basic syntax, so if you're an experienced Basic programmer, you'll be familiar with most of these commands already.

=LEFT$= *(Return the leftmost characters of a string)*

```
d$=LEFT$(s$,n)
LEFT$(d$,n)=s$
```

LEFT$ reads the first *n* characters to the left of string *s$* and copies them into the destination string *d$.*

As you can see, there are two general forms of this command. The first version is a function which creates a new destination string *d$* out of the first *n* characters from the source *s$.* Examples:

```
Print Left$(“AMOS Basic”,4)
AMOS
```

```
A$=Left$(“0123456789ABCDEF”,10)
Print A$
0123456789
```

```
Do
   Input “Input a String ?”;S$
   Input “Number of Characters ”;N
   Print Left$(S$,N)
Loop
```

The second form of LEFT$ replaces the leftmost *n* characters in the destination string with the equivalent characters in *s$.* Example:

```
A$=”**** Basic”
Left$(A$,4)=”AMOS”
Print A$
AMOS Basic
```

=RIGHT$= *(Return the rightmost character of a string)*

```
d$=RIGHT$(s$,n)
RIGHT$(d$,n)=s$
```

RIGHT$ copies *n* characters from *s$* to *d$* starting from the right. Examples:

```
Print Right$(“AMOS Basic”,5)
Basic
```

```
A$=Right$("0123456789ABCDEF",10)
Print A$
6789ABCDEF

Do
   Input "Input a string?";V$
   Input "Enter the number of characters?";N
   Print Right$(V$,N)
Loop
```

Like LEFT$ there's also a second version of RIGHT$ which is set up as a Basic instruction.

RIGHT$(d$,n)=s$

This loads the rightmost n characters of the source string *s$* into the destination string *d$*. Any excess characters in *s*$ will be totally ignored. Example:

```
A$="AMOS *****"
Right$(A$,5)="Basic"
Print A$
AMOS Basic
```

=MID$= *(Return a string of characters from within a string)*

d$=MID$(s$,p,n)
MID$(d$,p,n)=s$

The MID$ function returns the middle section of the string held in *s$*. *p* denotes the offset of characters to the start of this substring, and n holds the number of characters to be fetched. If a value of *n* is not specified in the instruction then the characters will be read right up to the end of your string. Examples:

```
Print Mid$("AMOS Basic",6)
Basic
Print Mid$("AMOS Basic",6,3)
Bas

Do
   Input "Input a string";V$
   Input "Enter the starting position, and the number of characters";S,N
   Print Mid$(V$,S,N)
Loop
```

There's also a MID$ instruction.

MID$(d$,p,n)=s$

This version of MID$ loads *n* characters into *d$* starting from position *p+1* in s$. If a value of *n* is not specified directly then the characters will be replaced up to the end of the source string *s$*. Examples:

```
A$="AMOS *****"
Mid$(A$,5)="Magic"
Print A$
AMOS Magic
Mid$(A$,5,3)="Bas"
Print A$
AMOS Basic
```

```
Do
   Input "Input a target string";V$
   Input "Input a substring";T$
   Input "Enter the starting position, and the number of characters";S,N
   Mid$(V$,S,N)=T$
   Print V$
Loop
```

=INSTR *(Search for occurrences of a string within another string)*

f=INSTR(d$,s$ [,p])

INSTR allows you to search for all occurrences of one string inside another. It is often used in adventure games to split a complete line of text into its individual commands. There are two possible forms of the INSTR function.

f=INSTR(d$,s$)

This searches for the first occurrence of s$ in d$. If the string is found then its position will be returned directly, otherwise the result will be set to zero. Examples:

```
Print Instr("AMOS Basic","AMOS")
1
Print Instr("AMOS Basic","S")
4
Print instr("AMOS Basic","AMIGA")
0
```

```
Do
   Input "String to be searched";D$
   Input "String to be found";S$
   X=Instr(D$,S$)
   If X=0 Then Print S$;" Not found"
   If X<>0 Then Print S$;" Found at position ";X
Loop
```

Normally the search will commence from the first character in your text string (*d$*). The second version of INSTR lets you test a specific section in the string at a time.

p is now the position of the beginning of your search. All characters are numbered from left to right starting from zero. Therefore *p* ranges from 0 to LEN(s$). Examples:

```
Print Instr("AMOS BASIC","S",0)
4
Print Instr("AMOS BASIC","S",5)
8
```

=UPPER$ *(Convert a string of text to upper case)*

s$=UPPER$(n$)

This function converts the string in *n$* into upper case (capitals) and places the result into *s$*. Example:

```
Print Upper$("AmOs BaSic")
AMOS BASIC
```

=LOWER$ *(Convert a string to lower case)*

s$=LOWER$(n$)

LOWER$ translates all the characters in *n$* into lower case. This is especially useful in adventure games, as you can convert all the user's input into a standard format which is much easier to interpret. Examples:

```
Print Lower$("AMOS Basic")
amos basic

Input "Continue (Yes/No)";ANSWER$
ANSWER$=Lower$(ANSWER$) : If ANSWER$="no" Then Edit
Print "Continuing with your program..."
```

=FLIP$ *(Invert a string)*

f$=FLIP$(n$)

FLIP$ simply reverses the order of the characters held in *n$*. Example:

```
Print Flip$("AMOS Basic")
cisaB SOMA
```

=SPACE$ *(Space out a string)*

s$=SPACE$(n)

Generates a string of *n* spaces and places them into *s$*. It's often used to pad out a piece of text before it's printed out onto the screen. Example:

```
Print "Twenty" ; Space$(20); "spaces"
```

=STRING$ *(Create a string full of a$)*

s$=STRING$(a$,n)

STRING$ returns a string with *n* copies of the first character in *a$*. Example:

```
Print String$("The cat sat on the mat",10)
TTTTTTTTTT
```

Note that STRING$(" ",N) is identical to SPACE$(N)

=CHR$ *(Return Ascii character)*

s$=CHR$(n)

Creates a string containing a single character with Ascii code n. Example:

```
For I=32 To 255 : Print Chr$(I); : Next I
```

Note that only the characters with codes 32 to 255 are actually printable on the screen. The rest are used internally as control codes. See text commands like CUP$ for more details.

=ASC *(Get Ascii code of a character)*

c=ASC(a$)

ASC supplies you with the internal Ascii code of the first character in the string *a$*. Example:

```
Print Asc("B")
66
```

=LEN *(Get length of string)*

l=LEN(a$)

LEN returns the number of characters stored in *a$*. Example:

```
Print Len("12345678")
8
```

Don't confuse this with the LENGTH function used to calculate the length of an AMOS memory bank.

=VAL *(Convert a string to a number)*

v=VAL(x$)
v#=VAL(x$)

VAL converts a list of decimal digits stored in x$ into a number. If this process fails for some reason, a value of zero will be returned instead. Example:

```
X=Val("1234") : Print X
1234
```

=STR$ *(Convert a number to a string)*

s$=STR$(n)

STR$ converts an integer variable into a string. This can be very useful because some functions, such as CENTRE, do not allow you to enter numbers as a parameter. Example:

```
Centre "Memory left Is "+Str$(Chip Free)+" Bytes"
```

Do not confuse STR$ with STRING$

Array operations

SORT *(Sort all elements in an array)*

SORT a(0)
SORT a#(0)
SORT a$(0)

The SORT instruction arranges the contents of any array into ascending order. This array can contain either strings, integers, or floating point numbers. The a$(0) parameter specifies the starting point of your table. It must always be set to the first item in the array (item number zero). Example:

```
Dim A(25)
P=0
Repeat
   Input "Input A Number (0 To Stop)";A(P)
   Inc P
Until A(P-1)=0 Or P>25
Sort A(0)
For I=0 To P-1
   Print A(I)
Next I
```

MATCH *(Search an array)*

r=MATCH(t(0),s)
r=MATCH(t#(0),s#)
r=MATCH(t$(0),s$)

MATCH searches through a sorted array for the value *s*. If this is successfully found then *r* will be loaded with the relevant index number. But if the search fails, the result will be negative. Taking the absolute value of this figure will provide you with the item which came closest to your original search parameter.

Note that only arrays with a single dimension can be checked in this way. You'll also need to sort the array with SORT before calling this function. Example:

```
Read N
Dim D$(N)
For I=1 To N
   Read D$(I)
Next I
Sort D$(0)
Do
   Input A$
   If A$="L"
      For I=1 To N : Print D$(I) : Next I
   Else
      POS=Match(D$(0),A$)
      If POS>0 Then Print "Found",D$(POS);" In Record ";POS
      If POS<0 And Abs(POS)<=N Then Print A$,"Not Found. Closest To ",D$(Abs(POS))
      If POS<0 And Abs(POS)>N Then Print A$,"Not Found. Closest To";D$(N)
   Endif
Loop
Data 10,"Adams","Asimov","Shaw","Heinlien","Zelazny","Foster","Niven"
Data "Harrison","Pratchet","Dickson"
```

Note that MATCH could be used in conjunction with the INSTR function to provide a powerful parser routine. This might be used to interpret the instructions you entered in an adventure game.

6: Graphics

AMOS Basic provides you with everything you need to generate some amazing graphics. There's a comprehensive set of commands for drawing rectangles, circles and polygons. As you would expect from the Amiga, all operations are performed practically instantaneously. But even here AMOS Basic has a trick or two up its sleeve.

The AMOS graphical functions work equally well in all the Amiga's graphics modes **including** hold and modify mode (HAM). It's therefore possible to create breathtaking HAM pictures directly within AMOS Basic!

Furthermore, you're not just limited to the visible screen. If you've created an extra large playing area, you'll be able to access every part of your display using the standard drawing routines. So it's easy to generate the scrolling backgrounds required by arcade games such as Defender.

Colours

The Amiga allows you to display up to 64 colours on the screen at a time. These colours can be selected using the INK, COLOUR and PALETTE commands.

INK *(Set colour used by drawing operations)*

INK col[,paper][,border]

col specifies the colour which is to be used for all subsequent drawing operations. The colour of every point on the screen is taken from one of 32 different *colour registers.*These registers can be individually set with a colour value chosen from a palette of 4096 colours.

Although the Amiga only provides you with 32 actual colour registers, AMOS lets you use colour numbers ranging from 0 to 63. This allows you to make full use of the colours available from the Half-Bright and HAM modes respectively. A detailed explanation of these modes can be found in the Screens chapter.

The *paper* colour sets the background colour fill patterns generated by the SET PATTERN command.

The *border* colour selects an outline colour for your bars and polygons. This option can be activated using the SET PAINT command like so:

```
Rem Draws boxes with random sizes at random positions
Set pattern 0 : Set Paint 1
Repeat
   C=Rnd(16) : Ink 16-C,0,C
   X=Rnd(320)-20 : Y=Rnd(200)-20 : S=Rnd(100)+10
   Bar X,Y To X+S,Y+S
 Until Mouse Key
```

Note that any of the parameters *col, paper* and *border* may be omitted. Simply include *empty* commas at the appropriate places in the instruction. For example:

```
Ink ,,5:Rem Just sets the border colour
```

COLOUR *(Assign a colour to an index)*

COLOUR index,$RGB

The COLOUR instruction allows you to assign a colour to each of the Amiga's 32 colour registers.

Index is the number of the colour you wish to change, and can range from 0-31. As you may know, any colour can be created by mixing specific amounts of the *primary* colours Red, Green and Blue. The shade of your colour is completely determined by the relative intensities of the three components.

The Amiga's hardware allows you to select each colour component from a range of 16 intensities. This can be used to generate 16x16x16 (4096) different colours. It's normal practice to specify these colours in hexadecimal format (base 16).

Hex digit	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The expression $RGB consists of three digits ranging from 0 to F. Each component sets the strength of one of the primary colours, Red (R), Green (G) or Blue (B). The size of the component is directly proportional to the brightness of the associated colour. So the higher the values, the brighter the eventual colour.

Here are a few examples of this notation:

Components	Hex form	Final Colour
R=0 G=0 B=0	$000	Black
R=F G=0 B=0	$F00	Bright red
R=8 G=0 B=0	$800	Dark red
R=F G=F B=0	$FF0	Yellow
R=0 G=F B=0	$0F0	Green
R=8 G=0 B=F	$F0F	Violet
R=F G=F B=F	$FFF	White
R=6 G=6 B=6	$666	Grey

So if you wanted to load colour number 5 with yellow, you would type:

```
Colour 5,$FF0
```

When this statement is executed, any graphics displayed on the screen which use colour number 5 will be immediately changed to the new colour. Note that HAM and Extra Half Bright modes use these indices slightly differently. See Chapter 9 for more details.

=COLOUR *(Read the colour assignment)*

c=COLOUR(index)

The COLOUR function takes an index number from 0 to 31, and returns the colour value which has been previously assigned to it.

index is simply the colour number whose shade you wish to determine. You can use this function to produce a list of the current colour settings of your Amiga like so:

```
For C=0 to 15
  Print Hex$(Colour(C),3)
Next C
```

PALETTE *(Set the current screen colours)*

PALETTE list of colours

The PALETTE instruction is really just a rather more powerful version of COLOUR. Instead of loading the colour values one at a time, the PALETTE command allows you to install a whole new palette of colours in a single statement.

However you don't have to set all the colours in the palette at once. Any combination of colours can be loaded individually, for example:

```
Palette $100,$200,$300 : Rem Sets just three colours
```

You can also change selected colours in the middle of your list like so:

```
Palette $200,,$400 : Rem Change colours 0 and 2
```

It's important to realise that only the colours in the palette which are specifically set by this command will actually be changed. All other colours will retain their original values. Here are some examples:

```
Palette 0,$F00,$0F0
Palette 0,$770
Palette 0,,$66
Palette 0,$1,$2,$3,$4,$5,$6,$7,$8,$8,$9,$A,$B,$C,$D,$E,$F
```

At the start of your program the colour palette is automatically loaded using a list of default colour values. These settings can be adjusted using a simple option from the AMOS configuration program.

This command can also be used to set the colours used by the Half-Bright and Ham modes. These extend the existing colour palette to generate dozens of extra colours on the screen. See Chapter 10 for a detailed explanation.

Line drawing commands

GR LOCATE *(Position graphics cursor)*

GR LOCATE x,y

This sets the position of the *graphics cursor* to screen coordinates *x,y*. The graphics cursor is used as the default starting point for most drawing operations. So if you omit the coordinates from commands such as PLOT or CIRCLE, the objects will be drawn at the current cursor position. For example:

```
Gr Locate 10,10 : Plot ,
Gr Locate 100,100 : Circle ,,100
```

=XGR *(Return X coordinate of graphics cursor)*
=YGR *(Return Y coordinate of graphics cursor)*

x=XGR
y=YGR

These functions return the present coordinates of the graphics cursor. For example:

```
Circle 10,100,100
Print Xgr,Ygr
```

PLOT *(Plot a single point)*

PLOT x,y [,c]

The PLOT command is the simplest drawing function provided by AMOS Basic. It plots a point at coordinates *x,y* using colour *c*. The new ink colour will now be used in all subsequent drawing operations.

If the colour *c* is omitted from this instruction, the point will be plotted in the current ink colour. For example:

```
Curs Off : Flash Off : Randomize Timer
Do
   Plot Rnd(319),Rnd(199),Rnd(15)
Loop
```

It's also possible to omit the X or Y coordinates from this instruction. The point will now be plotted at the graphic cursor position.

```
Plot 100,100,4
Plot ,150
Cls : Plot ,
```

POINT *(Get the colour of a point)*

c=POINT(x,y)

POINT returns the colour index of a point at coordinates *x,y*. For example:

```
Plot 100,100
Print "The colour at 100,100 is ";Point(100,100)
```

DRAW *(Draw a line)*

DRAW is another very basic instruction. Its action is to draw a simple straight line on the Amiga's screen.

DRAW x1,y1 TO x2,y2

Draws a line between the coordinates *x1,y1* and *x2,y2.*

DRAW TO x3,y3

Draws a line from the current graphics cursor to *x3,y3.* For example:

```
Colour 4,$707 : Ink 4
Draw 0,50 To 200,50
Draw To 100,100
Draw To 0,50
```

See also POLYLINE, INK.

BOX *(Draw a hollow rectangle on the screen)*

BOX x1,y1 TO x2,y2

The BOX command draws a hollow rectangular box on the Amiga's screen. *x1,y1* are the coordinates of the top left hand corner of the box, and *x2,y2* are the coordinates of the point diagonally opposite. For example:

```
Curs Off : Flash Off : Randomize Timer
Do
  Ink Rnd(15)
  X1=Rnd(320) : Y1=Rnd(200) : Box X1,Y1 To X1+Rnd(50),Y1+Rnd(50)
Loop
```

See also SET LINE, INK and BAR

POLYLINE *(Multiple line drawing)*

POLYLINE is very similar to DRAW except that it draws several lines at a time. It's capable of generating complex hollow polygons in just a single statement.

POLYLINE x1,y1 TO x2,y2 TO x3,y3 ...
POLYLINE TO x1,y1 TO x2,y2 ...

Where *x1,y1* = coordinates of point 1, *x2,y2* = point 2 and *x3,y3* = point 3

POLYLINE draws a line between each pair of coordinates in your list. So the first line is drawn from point 1 to point 2, the second from point 2 to point 3, and so on.

It's equivalent to the following lines:

```
Draw x1,y1 To x2,y2
Draw To x3,y3
Draw To x4,y4
```

Here's a simple example which draws a triangle on the Amiga's screen:

```
Polyline 0,20 To 200,20 To 100,100 To 0,20
```

Also see SET LINE, INK and POLYGON.

CIRCLE *(Draw a hollow circle)*

CIRCLE x,y,r

The CIRCLE command draws a hollow circle with radius *r* and centre *x,y*. For example:

```
Curs Off : Flash Off : Randomize Timer
Do
   Ink Rnd(15)
   X=Rnd(200) : Y=Rnd(100) : R=Rnd(90) : Circle X,Y,R
Loop
```

As normal, if the coordinates are omitted from this command, the circle will be drawn from the current cursor position. For example:

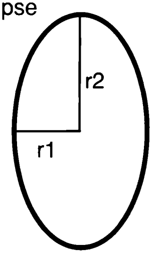
```
Plot 100,100 : Circle ,,50
```

ELLIPSE *(Draw a hollow ellipse)*

ELLIPSE x,y,r1,r2

The ELLIPSE instruction draws a hollow ellipse at coordinates *x,y*. *r1* is the horizontal radius. It corresponds to exactly half the width of the ellipse. *r2* is the vertical radius and is used to set the height of the ellipse. The total height of the ellipse is r*2.

The radii of an Ellipse



```
Curs Off : Flash Off : Randomize Timer
Do
   Ink Rnd(15) : X=Rnd(200) : Y=Rnd(100) : R1=rnd(90) : R2=rnd(90)
   Ellipse X,Y,R1,R2
Loop
```

Line types

AMOS Basic allows you to draw your lines using a vast range of possible line styles.

SET LINE *(Set the line styles)*

SET LINE mask

The SET LINE command sets the style of all lines which are subsequently drawn using the DRAW, BOX and POLYLINE commands.

Mask is a 16-bit binary number which describes the precise appearance of the line. Any points in the line which are to be displayed in the current ink colour are represented by a one, and any points which are to be set to the background colour are indicated by a zero. So a normal line is denoted by the binary number %1111111111111111 and will be displayed as: ________. Similarly, a dotted line like: _ _ _ _ will be produced by a mask of %1111000011110000.

By setting the line mask to values between 0 and 65535, it is possible to generate a great variety of different line types. For example:

```
Set Line $F0F0
Box 50,100 To 150,150
```

This line style is only applicable to straight lines, and has no effect on any shapes drawn with the CIRCLE or ELLIPSE commands.

Filled shapes

PAINT *(Contour fill)*

PAINT x, y, mode

The PAINT command allows you to fill any region on the Amiga's screen with a solid block of colour. Additionally you can select a fill pattern for your shapes using the SET PATTERN command.

x,y are the coordinates of a point inside the area to be filled. *mode* can be set to either 0 or 1. A value of 0 terminates the filling operation at the first pixel found with the current border colour.

A *mode* of 1 halts the filling operation at any colour which is different from the existing ink colour.

PAINT will happily fill any surface you like, providing it is completely enclosed by lines. However, if there is a gap in one of these lines, the fill colour will leak out into the rest of the screen. See **EXAMPLE 6.1** in the MANUAL folder for a demonstration.

BAR *(Draw a filled rectangle)*

BAR x1,y1 TO x2,y2

Draws a filled bar from *x1,y1* – the coordinates of the top left corner of the bar, to *x2,y2* – the coordinates of the corner diagonally opposite. For example:

```
Curs Off : Flash Off : Randomize Timer
Do
    X1=rnd(200) : Y1=rnd(100) : W=rnd(100) : H=rnd(80)
    Ink Rnd(15) : Bar X1,Y1 To X1+W,Y1+H
Loop
```

See also BOX, SET PAINT and INK

POLYGON *(Draw a filled polygon)*

POLYGON x1,y1 TO x2,y2 TO x3,y3 ...
POLYGON TO x1,y1 TO x2,y2 ...

POLYGON generates a filled polygon in the current ink colour. It's basically just a solid version of the standard POLYLINE command. As usual the fill colour can be set using the INK instruction, and the fill pattern with SET PAINT.

The coordinates *(x1,y1), (x2,y2),(x3,y3)* indicate the starting and ending points of the lines making up the polygon. There's no real limit to the number of coordinate pairs you may use, other than the maximum line length permitted by AMOS Basic (255 characters). This means you can create some very complicated shapes with this instruction.

There's also a second form of POLYGON which starts your polygon from the current cursor position. This has the format:

POLYGON TO x1,y2 TO x2,y2 ...

Apart from the starting coordinates, it's identical to the standard POLYGON instruction.

```
Do
    Ink Rnd(15)
    X1=Rnd(200) : Y1=Rnd(100) : H=Rnd(100) : W=Rnd(90)
    Polygon X1,Y1 To X1+W,Y1 To X1+W/2,Y1+H To X1,Y1
Loop
```

The program above fills the screen with pretty coloured triangles. Also see POLYLINE, INK, SET PAINT.

Fill types

In AMOS Basic you're not just restricted to filling your shapes with a solid block of colour. There are dozens of fill patterns to choose from, and you can even load your own patterns directly from the sprite bank.

SET PATTERN *(Select fill pattern)*

SET PATTERN pattern

This command allows you to select a fill pattern for use by your drawing operations.

There are three possibilities:

pattern=0

This is the default, and fills your shapes with a solid block of the current INK colour.

pattern>0

If the pattern number is greater than zero, AMOS Basic selects one of 34 built-in fill styles. These are found in the MOUSE.ABK file on your start-up disc, and can be edited using the AMOS Basic sprite definer. Note that the first three images in this file are required by the mouse cursor (see CHANGE MOUSE). The fill patterns are stored in the images from four onwards.

pattern<0

This is the most powerful option of all. *pattern* now refers to a sprite image in bank one. The image number is calculated using the formula:

SPRITE IMAGE = PATTERN *-1

The selected image will be automatically truncated before use, according to the following rules.

- The width of the image will be clipped to sixteen pixels
- The height will be rounded to the nearest power of two, ie 1, 2, 4, 8, 16, 32, 64.

Depending on the type of your image, the pattern will be drawn in one of two separate ways. Two–colour images are drawn in *monochrome.* The actual colours in your image are completely discarded, and the pattern is drawn using the current ink and paper colours.

It's also possible to produce *multi-coloured* fill patterns. In this case the foreground colours of your image are merged with the current ink colour using a logical AND. Similarly the paper colour of your pattern is OR'ed with the sprite background (colour zero). If you wish to use your original sprite colours, you'll need to set the ink and background colours like so:

```
Ink 31,0
```

Don't forget to load your sprite palette from the sprite bank with GET SPRITE PALETTE before using these instructions, otherwise the display is likely to look rather messy. Examples of this instruction can be found in **EXAMPLE 6.2** in the MANUAL folder. See CIRCLE, ELLIPSE, BAR and POLYGON.

SET PAINT *(Set / reset outline mode)*

SET PAINT n

The SET PAINT command toggles the outline drawn by the POLYGON or BAR instructions. As a default this mode set to OFF.

If *n=1* then outline mode will be activated, and a line will be drawn around your shape using the border colour specified in the previous INK command. For example:

```
Ink ,,5 : Set Paint 1 : Bar 100,100 to 200,150
```

You can turn the outlining off again by using SET PAINT with a value of zero.

Writing styles

GR WRITING *(Change writing mode)*

GR WRITING bitpattern

Whenever you draw some graphics on the Amiga's screen, you naturally assume that anything underneath it will be overwritten. The GR WRITING command allows you to choose from a range of four alternative drawing modes. These can be used to generate dozens of intriguing effects.

bitpattern holds a sequence of binary bits which specify which graphics mode you wish to use. Here's a list of the various possibilities along with a brief explanation of their effects:

JAM1 mode *Bit 0=0*
JAM1 only draws the parts of your graphics which are set to the current **ink** colour. Any sections drawn in the paper colour are totally omitted. This is particularly useful with the TEXT command as it allows you to merge your text directly over an existing screen background. For example:

```
Ink 2,5 : Text 140,80,"Normal Text" : Gr Writing 0 : Text 140,71,"JAM1"
```

JAM2 mode *Bit 0=1*
This is the default condition. Any existing graphics on the screen will be completely replaced by your new image.

XOR mode *Bit 1=1*
XOR combines your new graphics with those already on the screen using a logical operation known as eXclusive OR. The net result is to change the colour of the areas of a drawing which overlap an existing picture.

One interesting side effect of XOR mode is that you can erase any object from the screen by simply setting XOR mode and drawing your object again at exactly the same position. **EXAMPLE 6.3** contains a simple demonstration of this technique and produces a neat *rubber banding* effect.

INVERSEVID *Bit 2=1*
This reverses the image before it is drawn. So any sections of your image drawn in the ink colour will be replaced by the current paper colour and vice-versa. INVERSEVID mode is often used to produce inverted text.

Since these modes are set using a bitpattern, it's possible to combine several modes together.

```
Gr Writing 4+1 : Rem set JAM2 and INVERSEVID
Gr Writing 4+2+1: Rem chooses JAM2, INVERSVID and XOR
Ink 2,5 : Text 140,80,"Normal Text"
Gr Writing 5 : Text 140,71,"Inversevid+Jam2"
```

Note: This command only affects drawing operations such as CIRCLE, BOX and graphical text (TEXT). The drawing mode used by normal text commands like PRINT and CENTRE is set using a separate WRITING command. See also AUTOBACK and WRITING.

CLIP *(Restrict all graphics to a section of the screen)*

CLIP [x1,y1 TO x2,y2]

The CLIP instruction limits all drawing operations to a rectangular region of the screen specified by the coordinates *x1,y1* to *x2,y2.*

x1,y1 represent the coordinates of the top left hand corner of the rectangle, and *x2,y2* hold the coordinates of the bottom right corner.

Note that it's perfectly acceptable to use coordinates outside the normal screen boundaries. All the clipping operations will work as expected, even if only a section of the clipping rectangle is actually visible.

A detailed example of this command can be found in **EXAMPLE 6.4** in the MANUAL folder.

As you can see, only the parts of the circle which lie within the clipping rectangle have been drawn on the screen. The clipping zone can be restored to the normal screen area by omitting all the coordinates from this instruction.

Advanced techniques

SET TEMPRAS *(Set Temporary Raster)*

SET TEMPRAS [address,size]

This instruction allows experienced Amiga programmers to fine tune the amount of memory used by the various graphics operations. **Warning!** Improper use of this instruction can crash your Amiga completely!

Whenever an AMOS program performs a fill command, a special memory area is reserved to hold the fill pattern.This memory is automatically returned to the system after the instruction has terminated. The size of the memory buffer is equivalent to a single bit plane in the current screen mode. So the default screen takes up a total of 8k.

The size and location of the graphics buffer can be changed at any time using the SET

TEMPRAS instruction.

size is the number of bytes you wish to reserve for your buffer area. It ranges between *256 and 65536.*

The amount of memory required for a particular object can be calculated in the following way:

- Enclose the object to be drawn with a rectangular box.
- The area required will be given by: Size=Width/8*Height

If you are intending to use the PAINT command, you should take care to ensure that your figure is **closed**, otherwise more memory will be needed and the system may crash.

buffer can be either an address or a memory bank. The memory you reserve for this buffer should always be **chip ram**. Since the buffer area is now allocated once and for all at the start of your program, there's no need to continually reserve and restore the memory buffer. This can speed up the execution of your programs by up to 5%.

You can restore the buffer area to it's original value by calling the SET TEMPRAS command with no parameters.

See the **EXAMPLE 6.5** program on the MANUAL folder for a demonstration of this command.

7: Control structures

Most modern programming languages include a range of statements which allow you to make decisions and perform loops in your programs. These instructions are technically known as control structures. AMOS provides you with the full complement of Basic control structures. All your old favourites like GOTO and FOR...NEXT are supported, along with several intriguing new twists such as the ON...EVERY command.

GOTO *(Jump to a new line number)*

In the bad old days of computing, GOTO was probably the most commonly used of all the Basic instructions. Nowadays it's slightly unfashionable, and can be replaced by structured commands such as DO...LOOP and IF...ELSE...ENDIF. These are usually a great deal easier to read and you are recommended to avoid GOTO whenever possible.

The action of a GOTO is to transfer the control of the program one place to another. There are three forms of the GOTO command allowed in AMOS Basic.

GOTO label

label is an optional place marker at the side of a line. Label names are defined using the ":" colon character like so:

label:

The label name can consist of any string of alphanumeric characters you like, including "–". It's constructed using the same rules which apply for variables and procedure names.

GOTO line number

Any AMOS Basic line can be optionally preceded with a number. These *line numbers* are included solely for compatibility purposes with other versions of Basic (such as STOS for the Atari ST). It's better to rely on labels instead, as these are much easier to read and remember.

GOTO variable

Variable can be any allowable AMOS Basic expression. This expression may be either a normal integer or a string. Integers return a line number for your GOTO, whereas strings hold the name of a label.

Technically speaking, this construction is known as a computed goto. It's generally frowned upon by serious programmers, but it can be incredibly useful at times. Examples:

```
ROOM=3
BEGIN:
Goto “ROOM”+Str$(ROOM)-” “
```

```
End
ROOM3:
Print "Room three!"
Goto BEGIN
```

See also ON GOTO

GOSUB *(Jump to a subroutine)*

GOSUB is another outmoded instruction, and provides you with the useful ability to split a program into smaller, more manageable chunks, known as subroutines. Nowadays, GOSUB has been almost entirely supplanted by AMOS Basic's procedure system. However, GOSUB does form a useful half-way house when you're converting programs from another version of Basic such as STOS.

As with GOTO, there are three different forms of the GOSUB instruction.

GOSUB n	Jump to the subroutine at line *n.*
GOSUB name	Jump to an AMOS label.
GOSUB exp	Jump to a label or line which results from the expression in *exp.*

Example:

```
For I=1 To 10
   Gosub TEST
Next I
Direct
TEST:
Print "This is an example of GOSUB" : Print "I equals ";I
Return : Rem Exit from subroutine TEST and return to main program
```

It's good practice to always place your subroutines at the end of your main program as this makes them easier to pick out from your program listings. You should also add a statement like Edit or Direct to the end of your main program, as otherwise AMOS may attempt to execute your GOSUBs after the program has finished, generating an error message.

RETURN *(Return from a subroutine called by a GOSUB command)*

RETURN

RETURN exits from a subroutine which was previously entered using GOSUB. It immediately jumps back to the next Basic instruction after the original GOSUB.

Note that a single GOSUB statement can contain several RETURN commands. So you can exit from any number of different points in your routine depending on the situation.

POP *(Remove the RETURN information after a GOSUB)*

POP

Normally it's illegal to exit from a GOSUB statement using a standard GOTO. This can occasionally be inconvenient, especially if an error occurs, which makes it unacceptable to return to your program from the precise point you left it.

The POP instruction removes the return address generated by your GOSUB, and allows you to leave the subroutine in any way you like, without first having to execute the final RETURN statement. Example:

```
Do
  Gosub TEST
Loop
BYE:
Print "Popped out"
Direct : Rem keep subroutines separate from main program
TEST:
Print "Hi There!"
If Mouse Key Then Pop : Goto BYE
Return
```

See ON GOSUB

IF...THEN...[ELSE] *(Choose between alternative actions)*

The IF...THEN instruction allows you to make simple decisions within a Basic program. The format is:

IF conditions THEN statements1 [ELSE statements 2]

conditions can be any list of tests including AND and OR. *statements1* and *statements2* must be a list of AMOS Basic instructions. If you want to jump to a line number or a label, you'll have to include a separate GOTO command like so:

```
If test Then Goto Label : Rem this is fine
```

If you forget about this, and attempt to omit the GOTO statement as in normal Basic, AMOS will treat your Label as a procedure name, and you'll get a *procedure not defined* error.

```
If test Then Label : Rem This Calls a PROCEDURE
```

The scope of this IF...THEN statement is limited to just a single line of your Basic program. It has now been superceded by the much more powerful IF...ELSE...ENDIF command.

IF...[ELSE]...ENDIF *(Structured test)*

Although the original form of IF...THEN is undoubtedly useful, it's rather old fashioned when compared with the facilities found in a really modern version of Basic such as AMOS. This allows you to execute whole lists of instructions depending on the outcome of a single test.

```
IF tests=TRUE
    List of statements 1
    :     :     :
ELSE
    List of statements 2
    :     :     :
ENDIF
```

The list of statements can be any group of AMOS Basic instructions you wish, including other IF...ENDIF commands. However, it's **illegal** to use a normal IF...THEN inside a structured test. These should be replaced by their equivalent IF...ENDIF instruction like so:

```
If test Then Goto Label Else Label2
```

This now becomes:

```
If test : Goto Label : Else goto Label2 : Endif
```

or:

```
If test
   Goto Label
Endif
```

Here is an example of the IF...ENDIF statement in action:

```
Input "Enter values for a,b and c";A,B,C
If A=B
   Print "Equal"
Else
   Print "Different";
   If A<>B and A<>C
      Print ", and C Is Not The Same Too!"
   End If
End If
```

Each IF statement in your program **must** be paired with a single ENDIF command as this informs AMOS Basic precisely which group of instructions are to be executed inside your test.

Note that "THEN" is not used by this form of the instruction at all. This may take a little getting used to if you are already experienced with one of the other versions of Basic for the Commodore Amiga.

See AND, OR, NOT, TRUE, FALSE.

FOR...NEXT *(Repeat a section of code a specific number of times)*

This is the classic way of repeating sections of your Basic programs. The format of the instruction is:

```
FOR index=first TO last [STEP inc]
list of instructions
:  :  :
NEXT [index]
```

A FOR...NEXT command repeats your list of instructions a specific number of times.

index holds a counter which will be incremented after each and every loop. At the start of the loop, this counter will be loaded with the result of the expression in *first*.The instructions between the FOR and the NEXT are now performed until the NEXT is reached.

inc is a value which will be added to the counter after each loop by the NEXT instruction. If it's omitted, the increment will be automatically set to 1. After NEXT has updated this counter, it tests whether the current value is greater than *last*. If so, the loop is terminated immediately, and Basic executes the instruction straight after the NEXT. Otherwise the loop is restarted again from the top.

Note that if *inc* is negative, the loop will be halted when the counter is less than the value in *first*. So the entire loop will be performed in reverse.

Once inside your loop, *index* can be read from your program just like a normal variable. But you are **not** allowed to change its value in any way as this will generate an error message.

Each FOR statement in your program **must** be matched by a single NEXT instruction. You can't use the shorthand forms found in other Basics like NEXT R1,R1.Here are a couple of simple examples of these loops.

```
For I=32 To 255:Print Chr$(I); : Next I
```

```
For R1=20 To 100 Step 20
   For R2=20 To 100 Step 20
      For A=0 To 3
         Ink A
         Ellipse 160,100,R1,R2
      Next A
   Next R2
Next R1
```

See how we've placed a number of FOR...NEXT loops inside each other. This is known as nesting.

WHILE...WEND *(Repeat a section of code while a condition is true)*

The WHILE command provides you with a convenient method of repeating a series of Basic instructions until a certain condition has been satisfied.

```
WHILE condition
:      :
list of statements
:      :
WEND
```

condition can be any set of tests you like and can include the constructions AND, OR., and NOT. A check is made on each turn of the loop. If the test returns a value of -1 (true), then the statements between the WHILE and the WEND will be executed, otherwise the loop will be aborted and Basic will proceed to the next instruction.Type the following example:

```
Input "Type in a number";X
Print "Counting To 11"
While X<11
   Inc X
   Print X
Wend
Print "Loop terminated"
```

The number of times the WHILE loop in this program will be executed depends on the value you input to the routine. If you enter a number larger than 10, the loop will never be executed at all. WHILE will therefore only execute the statements if the condition is TRUE at the start of your program.

REPEAT...UNTIL *(Repeat until a condition is satisfied)*

```
REPEAT
:     :
list of statements
:     :
UNTIL condition
```

REPEAT...UNTIL is similar to WHILE...WEND except that the test for completion is made at the end of the loop rather than the beginning. The loop will be repeated continually until the specified condition is FALSE. So it will always be performed at least once in your program. Example:

```
Repeat
   Print "AMOS Basic"
Until Mouse Key<>0
```

As with WHILE...WEND you should always remember to match each REPEAT with an UNTIL.

DO...LOOP *(Loop forever)*

```
DO
:  :
List of statements
:  :
LOOP
```

The DO...LOOP commands take a list of Basic statements and repeat them continually. In order to exit from this loop, you'll need to use a special EXIT or EXIT IF instruction.

```
Do
   Print "Hi there"
Loop
```

The advantage of this system is that it's a structure alternative to the GOTO loops that tend to crop up in earlier versions of Basic. Take the following example:

```
TEST:
Input "Another game (Y,N)";AN$
If Upper$(AN$)="N" Then Goto BYE
GAME : Rem call play game procedure
Goto TEST
BYE:
End
```

This a fairly common type of routine but it's hardly easy to read. Now for a second version using DO...LOOP.

```
Do
   Input "Another game (Y,N)";AN$
   Exit If Upper$(AN$)="N"
   GAME : Rem call play game procedure
Loop
End
```

Hopefully, you'll agree that the new routine is much clearer.

EXIT *(Exit from a DO...LOOP)*

EXIT [n]

The EXIT command exits immediately from one or more program loops created with the FOR...NEXT, REPEAT...UNTIL, WHILE...WEND, or DO...LOOP statements. Your AMOS program will now jump directly to the next instruction after the current loop.

n is the number of loops you wish to leave. If it's omitted, then only the innermost loop will be terminated. Here's an example:

```
Do
   Do
      Print "Inner loop"
      If Mouse Key =1 Then Exit
      If Mouse Key=2 Then Exit 2
      Wait 5 : Rem Slow loop so you can see it running
   Loop
   Locate 20, : Print "Outer loop"
   Wait 5
Loop
```

EXIT IF *(Exit from a loop depending on a test)*

EXIT IF expression[,n]

As you can see from the previous EXIT instruction, it's often necessary to terminate a loop depending on the outcome of a specific set of conditions. This is simplified using a special EXIT IF command.

expression consists of a series of tests in the standard AMOS format. The EXIT will only be performed if the result evaluates to -1 (true).

The *n* parameter allows you to quit from several loops at the same time. If it's omitted then only the current loop will be aborted. Example:

```
While L=0
   Z=0
   Do
      Z=Z+1
      For X=0 To 100
         Exit If Z=10,2 : Rem Exits two Loops, DO and FOR
      Next X
   Loop
   Exit 1 : Rem Terminates While...Wend
Wend
```

EDIT *(Stop running the program and return to the Editor)*

EDIT

The EDIT directive stops the current program and returns to the AMOS Basic editor. This can be very useful when you are debugging one of your programs.

DIRECT *(Exit to direct mode)*

DIRECT

DIRECT terminates your program and jumps immediately back to the direct mode. You can now examine the contents of your variables or list your programs out to the printer.

END *(Exit from the program)*

END

This instruction exits from a program. You'll now be given the option to return to either the editor or to direct mode. Press SPACE to edit your program, or hit the ESCAPE key to jump to direct mode.

ON...PROC *(Jump to one of several procedures depending on a variable)*

ON v PROC proc1, proc2, proc3, ...procN

Jumps to a named procedure depending on the contents of variable *v*. Note that any procedures you use in this command **cannot** include parameters. If you need to transfer information to this procedure, you should place them in *global* variables instead. See PROCEDURES for a full explanation of this technique.

The On...PROC command is effectively equivalent to the following lines:

```
If v=1 Then Proc1
If v=2 Then Proc2
:   :   :
If v=n Then ProcN
```

ON...GOTO *(Jump to one of a list of lines depending on a variable)*

ON v GOTO line1,line2,line3...lineN

The ON GOTO instruction lets your program jump to one of a number of lines depending on the result of an expression in *v*. It's equivalent to the following lines:

```
If v=1 Then Goto Line1
If v=2 Then Goto Line2
:  :  :
If v=n Then Goto LineN
```

In order to have an effect, *n* must be a normal integer between 1 and the number of possible destinations. *line* may be either a line number or a label. See GOTO, GOSUB, ON GOSUB

ON...GOSUB *(GOSUB one of a list of routines depending on var)*

ON var GOSUB line1,line2,line3...

This is identical to ON...GOTO except that it uses a gosub rather than a goto to jump to the line. When the subroutine has finished executing, it should use a RETURN to jump back to the next instruction after the ON...GOSUB statement.

See also GOSUB and ON GOTO

EVERY n GOSUB *(Call a subroutine at regular intervals)*

EVERY n GOSUB label

The ON EVERY statement calls the subroutine at *label* at regular intervals, without interfering with your main program.

n is the length of your interval in 50ths of a second. The time taken for your subroutine to complete must always be less than this period, or you'll get an error.

After a subroutine has been entered, the system will be automatically disabled. In order to call this feature continuously, you'll therefore need to add an EVERY ON command before the final RETURN statement. Here's an example:

```
Every 50 Gosub TEST
Do
  Print "Main loop"
Loop
TEST:
Inc I : Print "This is call number ";I
Every On:Return
```

Note that ON EVERY is similar to the ON TIMER instruction in Amiga Basic.

EVERY n PROC *(Call a procedure at regular intervals)*

EVERY n PROC name

EVERY PROC executes the required procedure automatically at regular intervals using a powerful interrupt system.

n is the delay between each successive procedure call measured in units of a 50th of a second.

As with the previous command, the interrupt must be reactivated before leaving your procedure, otherwise the routine will only be called just once. So you'll need to use EVERY ON before returning to your main program with END PROC.

```
Every 50 Proc TEST
Do
  Print "Main loop"
Loop
Procedure TEST
  Shared I
  Inc I : Print "This is call number ";I
  Every On
End Proc
```

EVERY ON/OFF *(Toggle automatic procedure calls)*

EVERY ON/OFF

EVERY ON restarts the interrupt system used by the EVERY commands. It should be called just before the procedure or subroutine has finished executing.

Similarly EVERY OFF disables the calls completely. It's automatically executed at the start of one of these procedures.

BREAK ON/OFF *(Turn on or off the Control+C Break key)*

BREAK ON/OFF

Normally you can interrupt a program and return to Basic at any time by simply pressing the two keys Control and C. This function can be deactivated using the BREAK OFF command, providing your program with a crude form of copy protection. As you might expect, you can also restart the Break keys using BREAK ON.

But be warned: Never run a protected program unless you have made a backup copy on the disc first. Otherwise if the program gets stuck in a loop, you could easily end up losing several hours of your work.

Error handling

ON ERROR GOTO *(Trap an error within a Basic program)*

ON ERROR GOTO label

The ON ERROR command allows you to detect and correct the errors inside an AMOS Basic program, without having to return to the editor window. Sometimes, errors can arise in a program which are impossible to predict in advance. Take, for instance, the following routine:

```
Do
  Input "Enter two numbers";A,B
  Print A;" divided by ";B;" is ";A/B
Loop
```

This program works fine until you try to enter a zero for B. AMOS Basic will now attempt to divide A by zero which will give you a *division by zero* error.

You can avoid this problem by trapping the error with an ON ERROR GOTO instruction like so:

ON ERROR GOTO label

Whenever an error occurs in your Basic program, AMOS will now jump straight to *label*. This will be the starting point of your own error correction routine which can fix the error and safely return to your main program.

Note that the error handler **must** exit using a special RESUME instruction. You are not allowed to jump back to your program with a normal GOTO statement.

```
On Error Goto HELP
Do
```

```
   Input "Enter two numbers";A,B
   Print A;" divided by ";B;" is ";A/B
Loop
Rem Error Handler
HELP:
Print : Print : Bell
Print "I'm afraid you've attempted to divide"
Print "your number by zero."
Resume Next: Rem Return back to the next instruction
```

In order for this system to work, it's essential that an error does not arise inside your error correction routine, otherwise AMOS will halt your program ignominiously.

The action of ON ERROR GOTO can be disabled by calling ON ERROR with no parameters.

```
On Error : Rem Kill error traps
```

You can also use ON ERROR GOTO 0 for this purpose.

ON ERROR PROC *(Trap an error using a procedure)*

ON ERROR PROC name

Selects a procedure which will be called automatically if there's an error in the main program. It's really just a structured version of the previous ON ERROR GOTO statement.

Although your procedure must be terminated by an END PROC in the normal way, you'll need to return to your main program with an additional call to RESUME. This can be placed just before the final END PROC statement. Here's an example:

```
On Error Proc HELP
Do
   Input "Enter two numbers";A,B
   Print A;" divided by ";B;" is ";A/B
Loop
Rem Error Handler
Procedure HELP
   Print : Print : Bell
   Print "I'm afraid you've attempted to divide"
   Print "your number by zero."
   Resume Next : Rem Return back to Basic from the next instruction
End Proc
```

Your error handler can freely call any Basic procedure you like. Each routine can be safeguarded with its own individual error traps. However it's not possible to detect errors within the error handler itself. So if a problem occurs inside the HELP routine, AMOS will abort your program completely.

RESUME *(Resume execution of the program after an error)*

RESUME allows you to return to a section of Basic after an error handler which you've created with ON ERROR, has corrected the original problem.You should NEVER attempt to use GOTO in this context.

There are five possible formats of this instruction:

RESUME

Jumps back to the statement which caused the error and tries again.

RESUME NEXT

Returns to the instruction just after the one which generated the error.

RESUME line

Jumps to a specific line point in your main program. *line* can refer to either a label or a normal line number. This may NOT be used to re-enter a procedure!

Procedures are treated slightly differently. If you want to jump to a particular label, you have to place a special marker somewhere in the procedure you are checking for errors. This may be accomplished using the RESUME LABEL command. There are two separate versions.

RESUME LABEL label

Defines the label which is to be returned to after an error. This must be called outside your error handler just after the original ON ERROR PROC or ON ERROR GOTO statement.

RESUME LABEL

Used inside your error handler to jump straight back to the label you've set up with the previous command. Example:

```
On Error Proc HELP
Resume Label AFTER
Error 12
Print "Never printed"
AFTER : Print "I've returned here"
End
Procedure HELP
   Print "Oh Dear, I think there is an error!"
   Resume Label
```

```
Endproc
```

=ERRN *(Return number of the last error)*

e=ERRN

If you're creating your own error handling routines using the ON ERROR command, you'll need to be able to check precisely which error has occurred in the main program.

When an error occurs, ERRN is automatically loaded with its identification number. See the Appendix at the back of this manual for a full list of the possible errors.

```
Print Errn
```

ERROR *(Generate an error and return to the AMOS Editor)*

ERROR n

The action of the ERROR command is to actually generate an error. This may sound rather crazy, but it's often quite useful.

Supposing you have created a nice little error handling routine which is able to cope with all possible disc errors. ERROR provides you with a simple way of simulating all the various problems, without the inconvenience of the actual error. Example:

```
Error 40
```

Quits the program and prints out a *Label not defined* error. Another useful form of this instruction is:

```
Error Errn
```

This uses the ERRN function to print the current error condition after a problem in your program.

8: Text and windows

This chapter describes the text and windowing features supported by AMOS Basic. There are dozens of available commands which can be used to generate anything from a simple hi–score table to a fancy dialogue box.

Text attributes

PEN *(Set colour of text)*

PEN index

The PEN instruction sets the colour of all the text which will be displayed in the current window. This colour can be chosen from one of up to 64 different possibilities depending on the graphics mode you are using. For example:

```
For INDEX=0 To 15
Pen INDEX
   Print "Pen number ";INDEX;At(20,);"Colour"
Next INDEX
```

As a default the pen colour uses index number 2 (white).

=PEN$(n) *(Change the pen colour using control characters)*

a$=PEN$(n)

PEN$ returns a special control sequence which changes the pen colour inside a string. The new pen colour will be automatically assigned whenever this string is subsequently printed on the screen. For example:

```
C$=Pen$(2)+"White "+Pen$(6)+"Blue"
Print C$
```

The string returned by PEN$ is in the format: Chr$(27)+"P"+Chr$(48+n).

See COLOUR, PALETTE, PAPER.

PAPER *(Set colour of text background)*

PAPER index

PAPER chooses a colour for the background of your text. As with PEN, index must be a colour number between 0 and 63. For example:

```
Pen 2 : For INDEX=0 To 15
Paper INDEX : Print "Paper Number ";INDEX;Space$(10)
```

```
Next INDEX
```

The screen background normally defaults to colour number one (orange). See SCREEN OPEN for a list of the various other possibilities.

=PAPER$(n) *(Return a control sequence to set the paper colour)*

x$=PAPER$(index)

PAPER$ returns a character string which automatically changes the background colour when it is printed on the screen. For example:

```
Pen 1 : C$=Paper$(2)+"White "+Paper$(6)+"Blue"
Print C$
```

See PEN, COLOUR, PALETTE.

INVERSE ON/OFF *(Enter inverse mode)*

INVERSE ON
INVERSE OFF

The INVERSE command swaps the text and background colours set by the PEN and PAPER commands. This inverts the text which is printed in the current window.

INVERSE ON activates inverse printing. Similarly the mode can be turned off using a simple call to INVERSE OFF. For example:

```
Print "This Is Some Text In Normal Mode"
Inverse On : Print "This Is Some Inverted Text":Inverse Off
```

See SHADE, UNDER, WRITING.

SHADE ON/OFF *(Shade all subsequent text)*

SHADE ON
SHADE OFF

SHADE ON highlights your text by reducing the brightness of the characters with a mask pattern. The shade of your text can be returned to normal using SHADE OFF. Examples:

```
Shade On : Print "Shaded text"
Shade Off : Print "Normal text"
```

See UNDER, INVERSE, WRITING.

UNDER ON/OFF *(Set underline mode)*

UNDER ON
UNDER OFF

UNDER ON underlines your text when it's printed on the screen. Use UNDER OFF to turn off this mode. For example:

```
Under On :Print "Underlined"
Underlined
Under Off:Print "Normal"
Normal
```

See SHADE, INVERSE, WRITING.

WRITING *(Change text writing mode)*

WRITING w1 [,w2]

The WRITING command allows you to change the writing mode used for all subsequent text operations. This determines precisely how your new text will be combined with the existing screen data.

The first value chooses one of four writing modes:

w1=0	REPLACE (Default)	Your new text will obliterate anything underneath it.
w1=1	OR	Merges the characters onto the screen with a logical OR
w1=2	XOR	Characters are now combined with the screen using XOR
w1=3	AND	ANDs the new text with the screen background.
w1=4	IGNORE	All printing operations will now be completely ignored!

The second number chooses which parts of the text will be printed on the screen. This option can be omitted if required.

w2=0	Normal	The text is output to the screen along with its background.
w2=1	Paper	Only the background of the text is drawn on the screen.
w2=2	Pen	Ignores the paper colour and writes the text on a background of colour zero.

Do **not** confuse with GR WRITING.

Cursor functions

When you use the PRINT statement your characters will always be displayed at the current cursor position. AMOS includes a range of facilities which let you move this cursor to any part on the screen.

LOCATE *(Position the cursor)*

LOCATE x,y
LOCATE x,
LOCATE ,y

LOCATE moves the text cursor to the coordinates x,y. This sets the starting point for all future printing operations.

All screen positions are specified using a special set of *text coordinates*. These are measured in units of a single character relative to the top left corner of the text window. For instance the coordinates *15,10* refer to a point 10 characters down and 15 characters to the right.

The acceptable range of these coordinates will vary depending on the precise dimensions of your window and the size of your character set. If you attempt to print something outside these limits an error will be generated.

Note that the current screen is always treated as window 0. So you don't have to actually open a window before using one of these functions. For example:

```
Locate 10,10 : Print "Hi"
```

If you want to position the cursor on the current line, you can omit the Y coordinate completely. For example:

```
Print "Hi Score 10000"; : Locate 9, : Print "12345";
```

Similarly you can move the cursor vertically without affecting the existing X coordinate.

```
Clw : Locate ,10 : Print "Tenth Line";
```

See CMOVE, AT, XCURS, YCURS.

CMOVE *(Relative cursor movement)*

CMOVE w,h

CMOVE moves the cursor a fixed distance away from its present position. It works by adding the contents of the variables *w* and *h* to the current cursor coordinates. So if the cursor was at 10,10, then typing:

```
Cmove 5,-5
```

would move the cursor to 15,5.

Like LOCATE you can omit either one of the coordinates as required. For example:

```
Cmove ,2 : Rem Move the cursor two places down
Cmove 2, : Rem Move the cursor two places to the right
```

=AT *(Return a sequence of control characters to position the cursor)*

x$=AT(x,y)

The AT function allows you to change the position of text directly from inside a character string. It works by returning a string in the format:

```
Chr$(27)+”X”+Chr$(48+X)+Chr$(27)+”Y”+Chr$(48+Y)
```

Whenever this string is printed, the text cursor will be moved to the coordinates x,y. For example:

```
A$=”This”+At(10,10)+”Is”+At(1,2)+”The Power Of”+At(20,20) + “AMOS!”
Print A$
```

These AT commands are perfect for hi–score tables as they allow you to position your text once and for all during your programs initialisation phase. You can now update the score at the correct point on the screen using a single print statement. Here's an example:

```
HI_SCORE$=At(20,10)+”HI Score “
SCORE=1000
Print HI_SCORE$;SCORE
```

This is identical to the lines:

```
SCORE=1000
Locate 20,10 : Print “Hi Score”;SCORE
```

The first version is easier to change as you can move the high score table by editing just one string, no matter how may times it's used in your program.

See LOCATE, CMOVE, CUP$, CDOWN$, CLEFT$ and CRIGHT$.

Conversion functions

AMOS Basic provides you with four useful functions which readily enable you to convert between text and graphics coordinates.

=XTEXT *(Convert an X coordinate from graphic format to text format)*

t=XTEXT(x)

This function takes a normal X coordinate and converts it to a text coordinate relative to the current window. If the screen coordinate lies outside this window then a negative value will be returned. See **EXAMPLE 8.1**.

See YTEXT, LOCATE, WINDOPEN, XGRAPHIC, YGRAPHIC.

=YTEXT *(Convert a Y coordinate from a graphic format to text)*

t=YTEXT(y)

YTEXT converts a Y coordinate from the standard screen format into a text coordinate relative to the current window.

See XTEXT for more details. Also YGRAPHIC, XGRAPHIC, LOCATE.

=XGRAPHIC *(Convert an x coordinate from text to graphic format)*

g=XGRAPHIC(x)

The XGRAPHIC function is effectively the inverse of XTEXT in that it takes a text X-coordinate ranging from 0 to the width of the current window and converts it to an absolute screen coordinate. It's used to position text over an area of graphics on the screen. For a demonstration of this command see **EXAMPLE 8.2** in the MANUAL folder. See YGRAPHIC, XTEXT, YTEXT.

=YGRAPHIC *(Convert a y coordinate from text to graphic format)*

g=YGRAPHIC(y)

This function converts a text Y-coordinate into an absolute screen coordinate.

See XGRAPHIC, XTEXT, YTEXT.

Cursor commands

The text cursor serves as a visible starting point of all future text operations. It's usually displayed as a flashing horizontal bar, although this may be changed using the SET CURS and CURS OFF commands.

By moving the cursor on the screen, you can position your text practically anywhere you like. Remember, all coordinate measurements are taken using **text** coordinates relative to the current window.

AMOS provides you with dozens of simple commands which allow you to position the cursor precisely on the screen. It's also possible to change the physical shape of the text cursor directly from your Basic program.

HOME *(Cursor home)*

HOME

HOME moves the text cursor to the top left hand corner of the current window (coordinates 0,0). For example:

```
Clw : Rem Clear current window
```

```
Locate 10,10 : Rem Move cursor to 10,10
Print “A demonstration of “
Home : Print “Home” : Rem Move cursor to 0,0
```

See LOCATE, XCURS, YCURS.

CDOWN *(Cursor down)*

CDOWN

CDOWN pushes the text cursor down by a single line. Example:

```
Print “Example” : Cdown : Cdown : Print “of cdown”
```

=CDOWN$ *(Return a chr$(31) character)*

x$=CDOWN$

CDOWN$ is a function which returns a special control character which automatically moves the cursor when it is printed. So *Print CDOWN$* is identical to *CDOWN*. CDOWN$ allows you to combine several cursor movements in a single string. Occasionally this can be extremely useful. For example:

```
C$=”\”+Cdown$
For A=0 to 20
   Print C$;
Next A
```

See CUP, CLEFT, CRIGHT, AT.

CUP *(Cursor up)*

CUP

CUP moves the text cursor up a line in the same way that CDOWN shifts it down. For example:

```
Print “Example” : Cup : Cup : Print “of cup”
```

=CUP$ *(Return a chr$(30) character)*

x$=CUP$

CUP$ returns a control string which moves the cursor up by a single character. For example:

```
Print “The cursor jumps “+CUP$+”up a line...”
```

See CLEFT, CDOWN, CRIGHT, AT.

CLEFT *(Cursor left)*

CLEFT

The CLEFT instruction displaces the text cursor one character to the left. For example:

```
Print “Example” : Cleft : Cleft : Print “of cleft”
```

=CLEFT$ *(Control string for CLEFT chr$(29))*

x$=CLEFT$

The CLEFT$ function returns a control character which performs a CLEFT operation when it is printed. Example:

```
Print “Hello “;
Print Cleft$+Cleft$+”p “;
```

See CUP, CRIGHT, CDOWN, AT.

CRIGHT *(Cursor right)*

CRIGHT

CRIGHT is the exact opposite of CLEFT and moves the cursor one place to the right.

```
Print “Example” : Cright : Cright : Print “of cright”
```

=CRIGHT$ *(Generate a chr$(28) control string for CRIGHT)*

x$=CRIGHT$

CRIGHT$ returns a control string which performs a CRIGHT operation inside a text sequence. Example:

```
Print Cright$:Rem This has the same effect as CRIGHT
```

See CLEFT, CUP, CDOWN, AT.

XCURS *(Return the X coordinate of the text cursor)*

x=XCURS

XCURS is a variable containing the current X coordinate of the text cursor (in text format). Example:

```
Locate 10,0 : Print Xcurs
```

YCURS *(Return the Y coordinate of the cursor)*

```
y=YCURS
```

YCURS holds the Y coordinate of the text cursor (in text format).

SET CURS *(Set text cursor shape)*

```
SET CURS L1,L2,L3,L4,L5,L6,L7,L8
```

This instruction allows you to change the shape of the cursor to anything you like. The shape is specified by a list of bit-patterns held in the parameters *l1-l8.* Each parameter determines the appearance of one horizontal line of the cursor, numbered from top to bottom.

Every bit represents a single point in the current cursor line. If it's set to 1 then the point will be drawn using colour number 3 – otherwise it will be displayed in the current PAPER colour. The best way to familiarise yourself with this instruction is with an example.

```
L1=%11111111
L2=%11111110
L3=%11111100
L4=%11111000
L5=%11110000
L6=%11100000
L7=%11000000
L8=%10000000
Set Curs L1,L2,L3,L4,L5,L6,L7,L8
```

Normally the text cursor will be flashing continually. To remove this effect simply make a call to the FLASH OFF command before using this instruction.

CURS ON/OFF *(Enable/disable text cursor)*

```
CURS ON
CURS OFF
```

This command hides or re–displays the flashing cursor from the current window. It has no effect on the cursors used by any other window.

MEMORIZE X/Y *(Save the X or Y coordinate of the text cursor)*

```
MEMORIZE X
MEMORIZE Y
```

The MEMORIZE commands store the current cursor position in a safe place. You may now print any text on the screen you like without destroying the original cursor coordinates.

These may be reloaded using the REMEMBER commands.

REMEMBER X/Y *(Restore the X or Y coordinate of the text cursor)*

REMEMBER X
REMEMBER Y

REMEMBER positions the cursor at the coordinates saved by a previous call to MEMORIZE. If MEMORIZE has not been used then the appropriate coordinate will be set to zero.

An example of this command is included in the MANUAL folder under **EXAMPLE 8.3**.

CLINE *(Clear part or all of the current cursor line)*

CLINE [n]

Clears the line on which the cursor is positioned. If *n* is present then *n* characters are cleared starting at the current cursor position (without moving it).

Type in the following lines from the direct window:

```
Print "Testing Testing Testing";
Cmove -7,
Cline 7
Cline
```

CURS PEN *(Choose a new colour for the text cursor)*

CURS PEN n

Changes the colour of the text cursor to index number *n*. If your screen mode provides you with four or more colours then the cursor will default to colour three. This colour is animated using a flash sequence which is automatically assigned when AMOS is loaded. So if you choose a different colour, the cursor will be completely static. In order to produce a flashing cursor you would then need to define a new colour sequence using the FLASH command.

Also note that the new colour only applies to the currently open window. It has no effect on the cursors used by any other windows. Example:

```
Curs Pen 5
```

See FLASH, CURS ON/CURS OFF.

Text input/output

CENTRE *(Print a line of text centred on the screen)*

CENTRE a$

CENTRE takes a string of characters in *a$* and prints it in the centre of the screen. This text is always output on the current cursor line. For example:

```
Locate 0,1
Centre "This is a centered TITLE"
Cmove ,3
Centre "And this is another one"
```

=TAB$ *(Print tabulation)*

x$=TAB$

TAB$ returns a control character known as a TAB (Ascii 9). When this character is printed the text cursor will be immediately moved several places to the right. The size of this movement can be set using the SET TAB command. As a default, the tab spacing is set to four.

SET TAB *(Change the tabulation)*

SET TAB n

This specifies the distance the text cursor will move when the next TAB character is printed. For example:

```
Home : Rem Move cursor to coordinates 0,0
Set Tab 5 : Rem Set tab spacing To 5
Print Tab$;"Hi"; : Rem Prints Hi starting at 5,0
A$=Tab$+Tab$
Print A$;"There" : Rem Prints text at 15,0
```

See TAB$, CRIGHT.

REPEAT$ *(Repeat a string)*

x$=REPEAT$(a$,n)

The REPEAT$ function allows you to print out the same string of characters several times using a single PRINT statement.

It works by adding a sequence of control characters into variable X$. When this string is printed, AMOS simply repeats *a$* to the screen *n* times. Possible values for *n* range between 1 and 207. A full demonstration of this command can be found in **EXAMPLE 8.4**. The format of the control string is:

```
Chr$(27)+"RO"+A$+Chr$(27)+"R"+Chr$(48+n)
```

Advanced text commands

ZONE$ *(Set up a zone around a piece of text)*

x$=ZONE$(a$,n)

The ZONE$ function surrounds a section of text with a screen zone. After you've defined one of these zones you can check for collisions between the zone and the mouse using the ZONE function. This allows you to create powerful on-screen menus and dialogue boxes without having to resort to any complicated programming tricks.

a$* is a string containing the text for one of the "Buttons" in your dialogue box. This button will be activated automatically when you print *x$ to the screen.

n specifies the number of the screen zone to be defined. The maximum number of these zones depends on the value you previously specified with RESERVE ZONE.

See the **EXAMPLE 8.5** program in the MANUAL folder for a demonstration of this command. The format of the control string is:

```
Chr$(27)+"ZO"+A$+Chr$(27)+"R"+Chr$(48+n)
```

See ZONE, SET ZONE, RESERVE ZONE, RESET ZONE, BORDER$.

BORDER$ *(Add a border to some text)*

x$=BORDER$(a$,n)

This returns a string of control characters which instructs AMOS to draw a border around the required text. It's commonly used in conjunction with the ZONE$ command to produce the fancy buttons found in dialogue boxes and alert windows.

n is the border number ranging from 1 to 16 and *a$* holds the text to be enclosed by the border. The text in *a$* will start at the current cursor position so don't be surprised when you get strange results printing at 0,0. Example:

```
Locate 1,1 : Print Border$("AMOS Basic",1)
```

To create a screen zone surrounded by a border you would use a line like:

```
Print Border$(Zone$(" CLICK HERE ",1),2)
```

This would enclose the text with zone number 1 and border 2 The control sequence returned:

```
Chr$(27)+"EO"+A$+Chr$(27)+"R"+Chr$(48+n)
```

See ZONE$, ZONE, BORDER.

HSCROLL *(Horizontal text scrolling)*

HSCROLL n

This scrolls all the text in the currently open window horizontally by a single character position. *n* can take the following values:

1 = Move current line to the left
2 = Scrolls entire screen to the left
3 = Move current line to the right
4 = Move screen to the right

Don't confuse this command with the SCROLL instruction which moves the entire screen.

VSCROLL *(Vertical text scrolling)*

VSCROLL n

Scrolls the text in the currently open window vertically by a single character

1 = Any text at the cursor line and below is scrolled down.
2 = Text at cursor line or below is moved up.
3 = Only text from the top of the screen to the cursor line is scrolled up.
4 = Text from top of the screen to the current cursor position is scrolled down.

Blank lines are inserted to pad out the gap left by the scrolling operation.

Windows

The AMOS windowing commands allow you to restrict your text and graphics operations to just a part of the current screen.

AMOS windows can be used with the zone commands to produce effective dialogue boxes such as file selectors and high score tables. A typical warning box, for instance, can be easily generated with just a couple of lines of AMOS Basic.

WINDOPEN *(Create a window)*

WINDOPEN n, x ,y ,w ,h [,border [,set]]

The WINDOPEN instruction opens a window and displays it on the Amiga's screen. This window will now be used for all subsequent text operations.

n is the number of the window to be defined. AMOS allows you to create as many windows as you like, limited only by the amount of available memory. As a default, window number zero is assigned to the current screen. So don't attempt to re-open this window using WINDOPEN or change it with WIND SIZE or WIND MOVE.

x,y are the graphics coordinates of the top left hand corner of your new window. Since AMOS windows are drawn using the AMIGA's blitter chip, the window area must always lie on a 16-pixel boundary. In order to achieve this, the *x* coordinates are automatically rounded to the nearest multiple of 16. Additionally, if you've included a border for your window, the X coordinate will be incremented by a further eight. This will ensure that the working area of your window always starts at the correct screen boundary.There are no restrictions whatsoever on the *y* coordinates.

w,h specify the size in characters of the new window. These dimensions must always be divisible by 2.

Border selects a border style for your window. There are 16 possible styles, with values ranging between 1 and 16.

Window borders can also include up to two optional title lines. One title is displayed along the top of the window and another may be added at the bottom.

AMOS windows may contain either text or graphics, just like the Intuition system. Each window can be assigned it's own individual character set with the powerful WINDOW FONT command.There's also a powerful WIND SAVE instruction which saves the screen area inside your windows. Whenever you move one of these windows the contents underneath will be automatically redrawn. For example:

AMOS windows may contain either text or graphics, just like the Intuition system. There's also a powerful WIND SAVE command which saves the screen area inside your windows. Whenever you move one of these windows the contents underneath will be automatically redrawn. For example:

```
For W=1 To 3
   Windopen W,(W-1)*96,50,10,10,1
   Paper W+3 : Pen W+6 : Clw
   Print " Window ";W
Next W
```

You can flick between these windows using the WINDOW command. Try typing the following statements from direct mode.

```
Window 1 : Print "AMOS"
Window 3 : Print "in action!"
Window 2 : Print "Basic"
```

The active window can always be distinguished by a flashing cursor – though this can be turned off using the CURS OFF command if required.

WINDOW FONT *(Change window font)*

WINDOW FONT n

Changes the font used by the current window to set *n*. *n* is the number of a graphics font which has been previously installed with the GET FONT command. This font **must** have dimensions of exactly 8x8. Proportional fonts are not allowed.

Since the window borders make use of some of these characters, you may get rather odd results when you're using standard Workbench fonts.

WINDSAVE *(Save the contents of the current window)*

WINDSAVE

The WIND SAVE command allows you to move your windows anywhere on the screen without corrupting your existing display.

Once you've activated this feature, any windows you subsequently open will automatically save the entire contents of the windows underneath. This area will be redrawn whenever you close a window or move it to a new position.

It's important to note that this option saves the contents of the current window, rather than the one you are defining with WIND OPEN.

At the start of your program the current window will be the default screen and will take up a massive 32k of memory. If you wished to save the background underneath a dialogue box the most of this memory would be completely wasted.

The solution is to create a dummy window of the required size, and to position it over the zone you wish to save. You can now execute a WIND SAVE command and continue with your program as normal.

When you subsequently call up your dialogue box the area underneath will be saved as part of your dummy window. So it will be automatically restored after your box has been removed.

BORDER *(Change the window border of the current screen)*

BORDER n,paper,pen

The BORDER command sets the border of the current window to style number *n*. This border is drawn using a group of characters installed in the default font. It is therefore possible to create your own border styles using the font definer accessory.

The *paper* and *pen* options allow you to freely choose the colours of your border. Acceptable border numbers range from 1 to 16.

Any of the parameters may be omitted from this instruction so the following commands are all perfectly legal:

```
Border 2,,
Border 1,2,3
Border 2,,3 : Rem Don't forget to include the commas for any missing values
```

TITLE TOP *(Define the upper title for the current window)*

TITLE TOP t$

The TITLE TOP instruction sets the top line of the current window to the title string in *t$*. This title will now be displayed along the top border of your window. Only bordered windows may be titled in this way.

```
Windopen 5,1,1,20,10
Title Top "Window number 5"
Wait Key
```

TITLE BOTTOM *(Define the lower title for the current window)*

TITLE BOTTOM b$

This command assigns the string *b$* to the bottom title of the current window.

```
Windopen 5,1,1,20,10
Title Bottom "Window number 5"
Wait Key
```

WINDOW *(Change current window)*

WINDOW n

WINDOW activates window *n* as the current window. This window will now be used for all future text operations.

If the automatic saving system has been initiated, this window will be immediately redrawn along with any of its contents. Load **EXAMPLE 8.6** from the MANUAL folder for a demonstration.

=WINDON *(Return the value of the current window)*

w=WINDON

WINDON returns the identification number of the currently active window. Example:

```
Windopen Rnd(12)+1,10,10,10,10
Print "Window number ";windon," Activated"
```

WIND CLOSE *(Close the current window)*

WIND CLOSE

The WIND CLOSE command deletes the current window. If you've previously called the WIND SAVE command, this window will be replaced by the saved graphics, otherwise the area will be erased from the screen completely.

```
Windopen 1,1,8,38,18,1 : Print "Press a key to close this window"
Wait Key
Wind Close
```

WINDMOVE *(Move a window)*

WIND MOVE x,y

WINDMOVE moves the current window to graphics coordinates *x,y*. As with the original window definitions the *x* coordinate will be rounded to the nearest 16-pixel boundary. Examples:

```
Wind Save
Wind Open 1,0,2,30,10,1
Wind Save
For M=1 to 100
   Pen Rnd(15) : Paper(15) : Print : Centre “ Move the window! “
   Wind Move Rnd(30)+1,Rnd(100)+1
   Wait Vbl
Next M
```

```
Wind Save : Wind Open 1,0,2,10,5 : Wind Save
Curs Off : Paper 5 : Pen 0 : Clw
Print : Print “ Move the” : Print “ Mouse” : Print “ Pointer”
BEGIN:
On Error Goto STP
LOP:
WX=X Screen(X Mouse) : WY=Y Screen(Y Mouse)
If OX=WX and OY=WY Then Goto LOP Else Wind Move WX,WY
OX=WX : OY=WY : Goto LOP
STP:
OX=WX : OY=WY : Resume BEGIN
```

If window saving has been activated then this window will be redisplayed at the new position, otherwise the screen will be completely unchanged.

WINDSIZE *(Change the size of the current window)*

WIND SIZE sx,sy

This command changes the size of an AMOS window. The new sizes, *sx* and *sy,* are specified in units of a single character. *Sx* must be divisible by two. See **EXAMPLE 8.7**

If you've previously called the WIND SAVE command, the original contents of your window will be redrawn by this instruction. If the new window is smaller than the original one, any parts of the image which lie outside it will be lost. Alternatively, if you've expanded your window, the area around your saved region will be filled with the current paper colour. Also note that after a WINDSIZE command the text cursor is always reset to coordinates 0,0.

CLW *(Clear the current window)*

CLW

CLW erases the contents of the current window and fills it with a block of the present PAPER colour. For example:

```
Rem Clears window number W
Procedure CLEAR_WIN[W]
   WIND_OLD=Windon
   Window W : Clw
   Window WIND_OLD
End proc
```

Slider bars

AMOS incorporates three instructions which allow you to display a standard slider bar on the screen. These sliders cannot be manipulated directly with the mouse. In order to create a working slider bar, you'll need to write a small Basic routine to perform this operate in your main program. Due to the sheer power of the AMOS system, this is extremely easy to accomplish, and the results can be extremely impressive, as can be seen from **EXAMPLE 8.8**.

HSLIDER *(Draw a horizontal slider)*

HSLIDER x1,y1 TO x2,y2,total,pos,size

Draws a horizontal slider bar from *x1,y1* to *x2,y2. x1,y1* hold the coordinates of . the top left corner of the bar. *x2,y2* set the position of the point diagonally opposite.

total is the number of individual units which the slider will be divided into. Each unit represents a single item in the object you are controlling with the slider. So in the editor window, *total* would be set to the number of lines in the current program.

The size of each unit is calculated from the following formula.

(X2-X1)/TOTAL

pos is the position of the slider box from the start of the slider, measured in the units you specified using *total. size* is the length of the slider box in the previous units. Examples:

```
Hslider 10,10 to 100,20,100,20,5
Hslider 10,50 to 150,100,25,10,10
```

For a working demonstration of one of these sliders, load **EXAMPLE 8.9** from the manual folder.

VSLIDER *(Draw a vertical slider)*

VSLIDER x1,y1 TO x2,y2,total,pos,size

VSLIDER is almost identical to the previous HSLIDER instruction. It displays a simple slider from *x1,y1* to *x2,y2. x1,y1* and *x2,y2* set the position and size of your slider. Examples:

```
Vslider 10,10 To 20,100,100,20,5
Vslider 0,0 To 319,199,10,2,6
```

An additional example can be found in **EXAMPLE 8.10**. This provides you with a fully working vertical slider bar for you to examine.

SET SLIDER *(Sets the fill patterns used in a slider)*

SET SLIDER b1,b2,b3,pb,s1,s2,s3,ps

Although this command looks incredibly complicated, it's actually rather simple. SET SLIDER enters the colours and patterns to be used in the slider bars created with the HSLIDER and VSLIDER commands.

b1,b2,b3 set the ink, paper and outline colours for the background of the box. *pb* chooses the fill pattern to be used for these regions

Similarly, *s1,s2,s3* input the colours of the slider box, and *sp* selects the pattern it is to be filled with.

bp and *sp* can be any fill patterns you wish. ASs usual, negative value refer to a sprite image from the current sprite bank. This allows you to create amazing colorful slider boxes. Example:

```
Centre "<Press a key>" : Curs Off
Do
   B1=Rnd(15) : B2=Rnd(15) : B3=Rnd(15) : BP=Rnd(24)
   S1=Rnd(15) : S2=Rnd(15) : S3=Rnd(15) : SP=Rnd(24)
   Set Slider B1,B2,B3,BP,S1,S2,S3,SP
   Hslider 10,50 To 300,60,100,20,25
   Vslider 10,60 To 20,180,100,20,25
   Wait Key
Loop
```

Fonts

There are two different types of fonts available in AMOS – text fonts and graphic fonts. The text fonts are those used by the PRINT and WINDOW commands. Text fonts are known as character sets and each AMOS Basic window can have its own individual set. The graphic fonts are much more flexible and offer a wider range of styles:

Graphic text

Your Amiga computer is capable of displaying an impressive variety of different text styles. The original Workbench disc was supplied with eight attractive fonts in a range of sizes, and many more of these fonts are freely available from the public domain. If you've upgraded to Workbench 1.3 you'll also be able to design your own fonts using the FED program on the Extras disc.

AMOS Basic provides you with total support for these fonts. Text can be printed in any of the available typefaces at any point on the screen.

AMOS fonts can be used to add spice to even the most basic games. These are invaluable for producing the loading screens and hi-score tables in your games. So it's a good idea to make full use of them in your AMOS Basic programs.

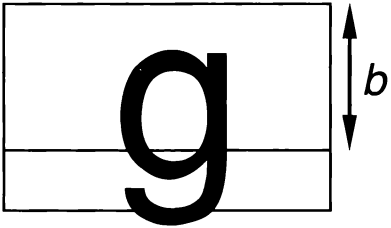
TEXT *(Print graphical text)*

TEXT x,y,t$

TEXT prints a line of text in *t$* at *graphical* coordinates *x,y*. All coordinates are measured relative to the characters *baseline*. This can be determined using a special TEXT BASE function.

Normally the baseline is positioned at the bottom of the character, but some lowercase letters – such as g – have a *tail* which extends slightly below this point, as shown in the following diagram:

The text base



As a default the type style is set to eight-point Topaz. This may be changed at any time using the SET FONT instruction. Try the following program and notice how text can be placed at any pixel position on the screen.

```
Do
  Ink Rnd(15)+1,Rnd(15): Text Rnd(320)+1,Rnd(198)+1,"AMOS Basic"
Loop
```

Also notice how the colour of your text is set with INK rather than the expected PEN and PAPER commands. This emphasizes the fact that the TEXT command is basically a graphical instruction. So the control sequences created by functions like CUP$ will be printed on the screen instead of being correctly interpreted.

There is no automatic line feed when the text reaches the end of the current window. If you attempt to print something too large, the text will be neatly clipped at the existing screen boundary. This can be seen by the example below.

```
Print String$("A",100):Text 0,100,String$("A",100)
```

GET FONTS *(Create list of all available fonts)*

GET FONTS

The GET FONTS command creates an internal list of all the fonts available from the current start-up disc. This list is essential to the running of the SET FONT command, so you should always call GET FONTS at least once before attempting to change the present font setting. The contents of this list can be examined using the FONT$ function.

Warning! In order for GET FONTS to work, your current AMOS work disc must always contain a copy of the standard LIBS folder along with its contents. It's important to remember this fact when you are distributing run-only or compiled programs because unless your discs contain the required files, AMOS Basic will almost certainly crash!

GET DISC FONTS *(Create a list of the DISC fonts)*

GET DISC FONTS

This command is identical to the previous GET FONTS instruction except that it only searches for fonts on the disc. These fonts are contained in the FONTS folder on your current boot-disc. If you want to use your own fonts with AMOS Basic, you'll need to copy these onto your normal start-up disc. See the manual supplied with your Amiga for details of this procedure.

GET ROM FONTS *(Create a list of the rom fonts)*

GET ROM FONTS

GET ROM FONTS produces a list of the fonts which are built into the Amiga's rom chips. At the present time there are just two of these fonts: Eight-point Topaz and nine-point Topaz.

```
Get Rom Fonts
Set Font 1
Text 100,100,"Topaz 9"
Set Font 2
Text 100,120,"Topaz 8"
```

=FONT$ *(Return details about the available fonts)*

a$=FONT$(n)

FONT$ returns a string of 38 characters which describes font number *n*. This function allows you to examine the font list created by a previous call to one of the GET FONT commands. *n* is the number of the font you wish to examine.

a$* contains a list of characters which hold the name and type of your font. If a font does not exist, *a$ will be loaded with a null value "", otherwise a string will be returned in the following format:

Character	Description
1-29	Font name
30-33	Font height
34-37	Identifier (set to either Disc or Rom)

An example of this command can be found in the file **EXAMPLE 8.11** in the MANUAL folder of your AMOS Program disc. This contains a number of useful procedures which may be freely used in your own programs.

SET FONT *(Choose a font for use by the TEXT instruction)*

SET FONT n

SET FONT changes the character set used by the TEXT command to font number *n*. If the font is stored on the disc it will be automatically loaded into your Amiga's memory. At the same time any previously sets which are not in use will be removed.

Here is a simple example of this command in action.

```
Get Fonts
Set Font 2 : Text 100,100,"AMOS" : Set Font 1 : Text 100,120,"Basic"
```

Notice how the GET FONTS instruction is called before selecting the font. This is because SET FONT makes use of the font list created with GET FONTS. A full demonstration of the SET FONT command can be found in **EXAMPLE 8.12**.

SET TEXT *(Set text style)*

SET TEXT style

The SET TEXT command allows you to change the style of a font. There are three styles to choose from: Underline, bold and italic.

style is a bit pattern in the following format:

Bit	Effect
0	Set this bit to one to underline your characters.
1	Selects **bold** characters.
2	Activates *italic* mode.

By setting the appropriate bits in this pattern you can choose between a total of eight different text styles. Here's an example for you to enter into your computer.

```
Colour 2,0 : Colour 1,$fff : Flash Off : Get Rom Fonts : Set Font 1
For S=0 To 7 : Set Text S : Text 100,S*20+20,"AMOS Basic" : Next S
```

=TEXT STYLE *(Return current text style)*

s=TEXT STYLE

This function returns the text style previously set from the SET TEXT command. The result in *s* is a bit-map in the same format as that used by SET TEXT.

=TEXT LENGTH *(Return the length of a section of graphic text)*

w=TEXT LENGTH(t$)

The TEXT LENGTH function returns the width in pixels of the character string *a$* in the current font. The width of a character varies depending on the size of your fonts. In addition, *proportional* fonts such as Helvetica assign different widths for each individual character. Here is a simple example:

```
T$="Centred Text"
L=Text Length(T$)
Text 160-L/2,100,T$
```

=TEXT BASE *(Return the current text base)*

b=TEXT BASE

This function returns the position of the *baseline* of your font. The baseline is the number of pixels between the top of a character and the point it will be printed on the screen. It's basically similar to the hot spot of a sprite of bob.

Installing new fonts

If you wish to use your own fonts within AMOS Basic, you'll need to install them onto a **copy** of your AMOS program disc. The basic procedure is as follows:

- Copy the required font files into the FONTS: directory of your boot disc.

- Further information can be found in the Extra's manual supplied with the Workbench 1.3 upgrade.

Troubleshooting

Although fonts are easy enough to use, there are still a couple of pitfalls for the unwary. Here's a list of the solutions to your more common problems.

Problem: GET FONTS seems to ignore any of the fonts on the current disc.
Solution: You've probably removed the original boot disc from your default drive. The Amiga's library routines expect to find the FONTS: directory on your start-up disc. This can be changed using the ASSIGN program in the UTILITIES folder. See GET DISC FONTS for more details.

Problem: GET FONTS crashes the Amiga completely.
Solution: This problem can easily occur when you're creating programs in run-only or compiled format. GET FONTS requires the DISCFONT library in the LIBS folder in order to work. If you forget to copy this folder onto your distribution discs you'll get a system error which may crash your Amiga.

Problem: The SET FONT command returns a *fonts not examined* error.
Solution: Add a call to GET FONTS to the start of your program. This will create a list of all the currently available fonts for use by the SET FONTS command.

9: Maths functions

AMOS Basic includes a wide variety of the more commonly needed mathematical functions.To conserve memory, AMOS uses the standard Amiga library routines. The appropriate libraries will be loaded automatically from your workbench disc the first time you call one of these functions in a particular session.You should therefore ensure that the current disc contains the file MATHTRANS.LIBRARY in the LIBS folder.

Trigonometric functions

The trigonometric functions provided you with a useful array of mathematical tools. These can be used for a variety of purposes, from education to the creation of complex musical waveforms.

DEGREE *(Use degrees)*

DEGREE

Generally all angles are specified in radians. Since radians are rather difficult to work with, it's possible to instruct AMOS to accept angles in degrees. Once you've activated this feature any subsequent calls to the trig functions will expect you to use degrees.

```
Degree
Print Sin(45)
```

RADIAN *(Use radian measure)*

RADIAN

The RADIAN directive informs AMOS that all future angles are to be entered using radians – this is the default.

=PI# *(A constant π)*

a#=PI#

This function returns the number called PI which represents the result of the division of the diameter of a circle by the circumference. PI is used by most of the trigonometric functions to calculate angles. Note that a # character is part of the token name. This is to avoid clashes with your own variable names.

=SIN *(Sine)*

s#=SIN(a)
s#=SIN(a#)

The SIN function calculates the sine of the angle in *a*. Note that this function always returns a floating point number. Example:

```
Degree
For X=0 To 319
   Y#=Sin(X)
   Plot X,Y#*50+100
Next X
```

See HSIN

=COS *(Cosine)*

c#=COS(a)
c#=COS(a#)

The cosine function computes the cosine of an angle. Normally all angles are measured in radians. This may be changed using the DEGREE command. Add the following two lines to the example above, ensuring they are inserted between the Plot and Next instructions.

```
Y#=Cos(X)
Plot X,Y#*50+100
```

See ACOS, HCOS

=TAN *(Tangent)*

t#=TAN(a)
t#=TAN(a#)

TAN generates the tangent of an angle. Examples:

```
Degree : Print Tan(45)
0.9999998
Radian : print Tan(Pi#/8)
0.4141
```

See ATAN, HTAN

=ACOS *(Arc cos)*

c#=ACOS(n#)

The ACOS function takes a number between -1 and +1 and calculates the angle which would be needed to generate this value with COS.

So if X#=COS(ANGLE) then ANGLE=ACOS(X#).

Note, we haven't provided you with ASIN, because it's not really needed. It can be readily computed using the formula:

```
ASIN(X)=90-ACOS(X) : Rem Measured in degrees
ASIN(X)=1.5708-ACOS(X) : Rem using Radians
```

Example:

```
A#=Cos(45)
Print Acos(A#)
```

See COS, HCOS

=ATAN *(Arc tangent)*

```
t#=ATAN(n#)
```

ATAN returns the arctan of a number. Example:

```
Degree : Print Tan(2)
0.03492082
Degree : Print Atan(0.03492082)
2
```

=HSIN *(Hyperbolic sine)*

```
s#=HSIN(a)
s#=HSIN(a#)
```

HSIN computes the hyperbolic sine of angle *a*.

=HCOS *(Hyperbolic cosine)*

```
c#=HCOS(a)
c#=HCOS(a#)
```

HCOS calculates the hyperbolic cosine of angle *a*.

=HTAN *(Hyperbolic tangent)*

```
t#=HTAN(a)
t#=HTAN(a#)
```

HTAN returns the hyperbolic tangent of the angle *a*.

Standard mathematical functions

=LOG *(Logarithm)*

```
r#=LOG(v)
r#=LOG(v#)
```

LOG returns the logarithm in base 10 (log10) of the expression in *v#*.
Examples:

```
print Log(10)
V#=Log(100)
```

=EXP *(Exponential function)*

```
r#=EXP(e#)
```

Calculates the exponential of e#.
Examples:

```
Print Exp(1)
2.71828
```

=LN *(Natural logarithm)*

```
r#=LN(l#)
```

LN computes the natural or naperian logarithm of l#. Examples:

```
Print Ln(10)
2.30258
R#=Ln(100) : Print R#
4.60517
```

=SQR *(Square root)*

```
s#=SQR(v)
s#=SQR(v#)
```

SQR calculates the square root of a number. Example:

```
Print Sqr(9)
3
Print Sqr(11.1111)
3.33333
```

=ABS *(Absolute value)*

r=ABS(v)
r#=ABS(v#)

ABS returns the absolute value of *v*, taking no account of its sign. .Example:

```
Print Abs(-1),Abs(1)
1 1
```

=INT *(Convert floating point number to an integer)*

i=INT(v#)

INT rounds a floating point number in *v* down to the nearest whole integer. Examples:

```
Print Int(1.25)
1
Print Int(-1.25)
-2
```

=SGN *(Find the sign of a number)*

s=SGN(v)
s=SGN(v#)

SGN returns a value representing the sign of a number. There are three possibilities:

-1 if *v* is negative
0 if *v* is zero
1 if *v* is positive

Creating random sequences

=RND *(Random number generator)*

v=RND(n)

RND generates a random integer between 0 and *n* inclusive. But If *n* is less than zero, RND will return the last value it produced. This can be very useful when debugging one of your programs.
Examples:

```
Do
   C=Rnd(15) : X=Rnd(320) : Y=Rnd(200) : Ink c : Text X,Y, " RANDOM "
Loop
```

RANDOMIZE *(Set the seed of a random number)*

RANDOMIZE seed

In practice, the numbers produced by the RND function are not really random. They're computed internally using a complex mathematical formula. The starting point for this calculation is taken from a number known as the *seed.* This seed is set to a standard value whenever you load AMOS Basic into the computer. So the sequence of numbers generated by RND will be exactly the same every time you run your game!

Although this may be acceptable for arcade games, it would obviously cause serious problems if you wanted to simulate a card game such as Poker.

The RANDOMIZE command allows you to solve this problem by setting the value of the seed directly.

seed can be any value you wish. Each seed generates its own individual sequence of numbers.

In order to generate true random numbers, you need some way of varying the seed from game to game. This can be achieved using the TIMER instruction:

```
Randomize Timer
```

TIMER is a Basic function which returns the amount of time which has elapsed since your Amiga was switched on in the current session. All timings are measured in units of a 50th of a second.

The best place to use this instruction is just after the user has entered some information into the computer. Even something simple as waiting for a keypress before starting the game will suffice, as it's obviously impossible to predict to the nearest 50th of a second when the user will hit a particular key.

Providing the contents of TIMER are reasonably changeable, the sequence of numbers produced by the RND function will be different in every game.

Manipulating numbers

=MAX *(Get the maximum of two values)*

r=MAX(x,y)
r#=MAX(x#,y#)
r$=MAX(x$,y$)

MAX compares two expressions and returns the largest. These expressions can be composed of numbers or strings of characters, providing you don't try to mix different types of expressions in one instruction.

```
Print Max(10,4)
10
Print Max("Hello","Hi")
Hi
```

=MIN *(Return the minimum of two values)*

r=MIN(x,y)
r#=MIN(x#,y#)
r$=MIN(x$,y$)

The MIN function returns the smallest value of two expressions. These may consist of strings, integers or real numbers. However you must only compare values of the same type. Examples:

```
A=Min(10,4) : Print A
4
Print Min("Hello","Hi")
Hello
```

SWAP *(Swap the contents of two variables)*

SWAP x,y
SWAP x#,y#
SWAP x$,y$

Swaps the data between any two variables of the same type. It's equivalent to a line like:

```
DUMMY=X : X=Y : Y=DUMMY
```

Example:

```
A=10 : B=40 : Swap A,B : Print A,B
```

FIX *(Set precision of floating point output)*

FIX(n)

FIX changes the way your floating point numbers will be displayed on the screen or printer. There are four possibilities.

If 0<*n*<16 then *n* denotes the number of figures to be output after the decimal point.

If *n*>16 the printout will be proportional and any trailing zeros will be removed.
If n=16 then the format will be returned to normal.

If *n*<0 then all floating point numbers will be displayed in exponential format, and the absolute value of *n* (ABS(n)) will determine the number of digits after the decimal point. Examples:

```
Fix (2) : Print PI# : Rem Limits the number to two digits after the point.
Fix (-4): Print PI# : Rem Forces exponential mode with four figures after the point.
Fix (8) : Print PI # : Rem Reverts to the normal mode.
```

Warning: This function is completely different from the equivalent command in AMIGA Basic .

DEF FN *(Create a user-defined function)*

DEF FN name [(list)]=expression

The DEF FN command lets you create your own user-defined functions within an AMOS Basic program. These can be used to compute commonly needed values quickly and easily.

name is the name of the function you wish to define. *list* is a set of variables separated by commas. Only the type of these variables is significant. When you call your function, any variables you enter with, will be automatically substituted in the appropriate positions.

expression can include any of the standard AMOS functions you wish. Like all Basic expressions, it's limited to just a single line of your program.

The new function can be called using the FN statement.

FN *(Call a user-defined function)*

FN name [(variable list)]

FN executes a function defined using DEF FN. Here are a couple of simple examples:

```
Def Fn Asin(X)=90-Acos(X)
Degree
Print Fn Asin(0.5)
30

Def Fn SLICE (A$,X,Y)=Mid$(A$,X,Y)
Print Fn SLICE("Hello",2,3)
ell
```

See how we've defined the function with DEF FN before we used it in our program. This is essential.

10: Screens

Your Commodore Amiga is capable of displaying some truly breathtaking pictures. AMOS Basic allows you to incorporate these images directly into your games with an impressive range of screen animation commands.

Each graphics mode is treated in exactly the same way, so it's as easy to create a 4096-colour Ham screen as it is to produce a standard 16-colour display. It's also possible to link two separate screens together using the dual playfield system. This can be exploited to produce amazing parallax effects similar to those in top commercial games such as Xenon II or Silkworm!

The default screen

Whenever you run an AMOS Basic program a default screen is created as screen zero. This forms a standard display which will be used for all your normal drawing operations.

The system defaults to a 16-colour screen with dimensions 320x200, which can easily be altered from within your program. In addition, you can also define up to seven further screens with the powerful SCREEN OPEN command.

Defining a screen

SCREEN OPEN *(Open a screen)*

SCREEN OPEN n,w, h, nc, mode

SCREEN OPEN opens a screen, and reserves some memory for it. The new screen will now be used as the destination of all subsequent text and graphical operations in your program.

n is the identification number of the screen which is to be created by this instruction. Possible values range from 0-7. If this screen already exists, it will be totally replaced by your new definition.

w holds the width of the screen in pixels. This is not limited to the physical size of your display. It's perfectly legal to define extra large screens which may be manipulated using SCREEN OFFSET.

h sets the height of your screen using the same system. Providing you've enough memory, you can easily create screens which are much larger than the visible screen area. These screens can be used in conjunction with all the normal screen operations. So you can construct your images off-screen, and scroll them into view with the SCREEN OFFSET command.

nc requests the number of colours required for the new screen. The range of available colours varies from 2 and 64 (Extra Half Bright). You can also access the Amiga's special Hold and Modify (Ham) mode with a value of 4096.

mode allows you to choose the width of the individual points on the screen. The Amiga supports screen widths of either 320 or 640 pixels. You can select the required width by setting *Mode* to either LOWRES (0) or HIRES ($8000).

Here's a list of the possible screen options along with an indication of the amount of memory they consume.

Colours	Resolution	Memory	Notes
2	320x200	8k	PAPER=0 PEN=1 Cursor=1, no flash
	640x200	16k	PAPER=0 PEN=1 Cursor=1, no flash
4	320x200	16k	PAPER=1 PEN=2 Cursor=3, flash=3
	640x200	32k	: : :
8	320x200	24k	PAPER=1 PEN=2 Cursor=3, flash=3
	640x200	48k	
16	320x200	32k	This is used by screen 0 (default).
	640x200	64k	
32	320x200	40k	
64	320x200	48k	Extra Half bright mode
4096	320x200	48k	Hold and modify mode

Note that the memory sizes in the table only apply to a standard screen. If you create taller or wider screens, the amount of memory which is consumed will obviously be considerably greater. Screen zero is equivalent to:

```
Screen Open 0,320,200,16,Lowres
```

Here are some more examples of AMOS screens.

```
Rem Opens a 640x200 hires screen with 8 colours
Screen open 1,640,200,8,Hires

Rem Opens screen 2 as a HAM screen
Screen open 2,320,256,4096,Lowres

Rem Opens screen 3 as a large 8 colour screen.
Rem Only the first 320x256 will be visible at any one time.
Screen open 3,500,400,8,Lowres

Rem This demo opens 8 screens and prints to all of them!
Curs Off : Cls 13 : Paper 13
Print : Centre “I'm screen 0 at the back!”
For A=1 To 7
    Screen Open A,320,20,16,Lowres
    Curs Off
    Cls A+5
    Paper A+5
    Centre “I'm screen “+Str$(A)
    Screen Display A,,50+A*25,,8
Next A
Direct
```

See SCREEN OFFSET, SCREEN DISPLAY and VIEW

SCREEN CLOSE *(Erase a screen)*

SCREEN CLOSE n

SCREEN CLOSE deletes screen number *n,* and frees the memory area it uses for the rest of your program.

AUTO VIEW ON/OFF *(Control viewing mode)*

AUTO VIEW OFF

When you open a screen using SCREEN OPEN the new screen is usually displayed immediately. This can be very inconvenient during the initialisation stages of your programs.

The AUTO VIEW OFF command provides you with full control over the updating process. It turns off the automatic display system completely. You can then update the screen display at a convenient point in your program using the VIEW instruction.

AUTO VIEW ON

Activates automatic screen updating. In this mode, any screen change will be immediately displayed on your TV set.

DEFAULT *(Reset screen to its default)*

DEFAULT

Closes all current open screens and restores the display back to it's original default setting. Example:

```
Load Iff "AMOS_DATA:IFF/AMOSPIC.IFF",0
Wait Key
Default
```

VIEW *(Display the current screen settings)*

VIEW

VIEW displays any changes to the current screen settings at the next vertical blank period. You only have to use this command when AUTOVIEW is OFF.

Special screen modes

The colour of every point on the screen is determined by a value held in one of the Amiga's 32 colour registers. Each register can be loaded from a selection of 4096 different colours.

Although 32 colours may seem rather a lot, particularly by ST standards, it wasn't enough for the Amiga's designers. The easiest solution would have been to increase the number of colour registers, but this was quickly ruled out for reasons of cost.

Instead, they invented two special graphics modes which cleverly exploited the existing registers to increase the maximum number of colours on the screen.

You've probably encountered these modes already. They're the infamous Extra Half Bright and Ham modes. AMOS Basic provides full support for both Ham and Half Bright modes. Here's a brief explanation.

Extra Half Bright mode (EHB)

Extra Half Bright mode doubles the maximum colours on the screen to a grand total of 64. It works by generating two colours for each of the 32 possible colour registers.

The first 32 colours load the colour value directly from one of the registers. Each register contains a value between 0 and 4095 which sets the precise shade of the final colour.

The second group of colours, with numbers from 32 to 63, take one of the previous registers and divide its contents by two. This produces 32 extra colours which are exactly half as bright as the normal colour registers.

Supposing colour zero contained a value of $FFF (White). Colour number 32 would now be displayed using a value of $777 (Grey). The shade of the colours from 32 to 63 can be worked out using the following simple formula:

```
colour n=(colour n–32)/2
```

Here's an example which demonstrates this principle:

```
Screen Close 0
Screen Open 2,320,167,64,Lowres : Flash Off
For I=1 To 32
   Ink I
   Bar 0,(I-1)*5 To 160,(2+I-1)*5
   Ink I+32
   Bar 160,(I-1)*5 To 319,(2+I-1)*5
Next I
```

In order to exploit EHB mode to the full, it's necessary to load the 32 registers with the brightest shades in your palette. This will automatically generate a list of intermediate tones in colours 32-63.

Aside from the colour palette, EHB screens are identical to any other screen mods. There are no restrictions whatsoever to their use. It's even possible to create Bobs in this mode!

Hold And Modify mode (HAM)

The Amiga's hardware currently limits you to a maximum of six bit planes per screen. This allows your to display up to 64 different colours on the screen at once. If you wanted to display a photograph though, you'd require hundreds or even thousands of colours on the screen.

This was the problem faced by Jay Miner when he was designing the Amiga's display system. His solution was to exploit a trick which has been known by artists for centuries.

If a professional artist had to take every conceivable colour on an assignment, he would be faced with an impossible task. It's therefore common practice to mix the exact shade on the spot, out of a small set of basic colours. This provides millions of potential shades, without the need to carry several large lorry loads worth of paint.

The same technique can also be applied to a computer screen. Instead of specifying each colour individually, you can take an existing colour and modify it slightly. This increases the number of available colours tremendously, and forms the basis of the Amiga's powerful **H**old **A**nd **M**odify mode (Ham).

Each colour value on the Amiga is created from a mixture of three separate components. These determine the relative strength of the primary colours Red, Green, and Blue in the final colour. Possible intensities range from 0 to 15.

Ham mode splits the Amiga's colour values into four separate groups:

- **Colour registers 0-15:** The first 16 colours take a value directly from a colour register. These colours are treated just like those on a standard 16 colour screen.

- **Red Components 16-31:** However, if a point is set to a colour number in the range 16 to 31, the colour value is loaded from the pixel to its immediate left. The Red component of this colour is now replaced with a value from 0 to 15 which is calculated from the formula:

  Intensity=Colour index-16

- **Green components 32-47:** Similarly, a colour number from 32 to 47 takes the current shade, and changes the green component. The intensity of this component is set to a value of Colour-32.

- **Blue components 48-63:** These colour numbers grab the colour value from the point on the left of the current pixel, and load a new blue component from your colour number like so:

  Intensity=Colour Index-48

The colour of a particular point therefore depends on the colours of all the points to the left of it. This allows you to create smooth gradations of colour which are ideal for flesh tones. However, you can't choose the colour of each point on the screen independently. In practice, it takes a maximum of three pixels to shift from one colour to another.

When the Amiga was first released, Ham initially was regarded as little more than curiosity. Nowadays, the situation is very different, with the advent of excellent Ham graphics packages such as Photon Paint.

AMOS allows you to perform the full range text and graphics operations directly on to a Ham screen. **Example 10.1** provides you with a simple example of how you can generate an entire screen in just a few lines of Basic code.

Another point to consider, is that Ham screens can be manipulated using the normal SCREEN DISPLAY and SCREEN OFFSET commands. Here are some simple guidelines to their use:

- The first point in each horizontal line should be set to a colour number from 0 to 15. This will serve as the starting colour for all the shades on the current line.

- Don't attempt to subject your Ham screens to horizontal scrolling. If you try to scroll one of these screens, you'll get colour fringes at the sides of your picture. These are generated by the changes in the starting colours for each line. There are no such restrictions to vertical scrolling.

- Fringing effects can also be produced by SCREEN COPY. The solution is to ensure that the border of your zone is drawn using a colour from 0 to 15. This will ensure that your Ham screens will be redrawn at their new position with their original colours.

Loading a screen

LOAD IFF *(Load an IFF screen from the disc)*

LOAD IFF "filename"[,screen]

LOAD IFF loads an IFF format picture from the disc. IFF format is now supported by the vast majority of the Amiga's drawing packages, so you should have little difficulty in loading own your artwork directly into AMOS Basic.

screen indicates the number of the screen which is to be loaded with your picture. This screen will be opened automatically for your use. If it already exists, anything inside it will be erased competely.

To load the picture into the present screen, omit the *screen* parameter altogether. Example:

```
Load Iff "AMOS_DATA:IFF/AMOSPIC.IFF",1
```

Saving a screen

SAVE IFF *(Save an IFF screen)*

SAVE IFF "filename"[,compression]

SAVE IFF saves the current screen as an IFF picture file on the disc. *compression* is a flag which allows you to choose whether your file will be compacted before it is saved. A value of one specifies that the standard file compression system is to be employed and zero saves the picture as it stands. As a default all AMOS screens are **compressed**.

SAVE IFF automatically appends a small IFF "chunk" to your picture file. This stores the present screen settings including SCREEN DISPLAY, SCREEN OFFSET and SCREEN HIDE/SHOW. When you load this file back into AMOS Basic it will be returned to exactly its original condition. This extra IFF data will be completely ignored by external graphics packages such as DPaint 3.

Note that it's not possible to save double buffered or dual playfield screens with this command.

Moving a screen

SCREEN DISPLAY *(Position a screen)*

SCREEN DISPLAY n [, x, y, w,h]

Once you have defined your screen with SCREEN OPEN, you'll need to position it on your screen. Unlike most other computers, the Amiga is capable of displaying a picture anywhere you like on the TV screen. This can be easily exploited to produce amazing "bouncing" screen effects. With AMOS Basic, it's even possible to perform these animations using interrupts (see AMAL).

Another application is to overlay several screens alongside each other. This allows you to create your display out of a combination of different screen modes.

n indicates the number of the screen to be positioned. *x* and *y* specify the location of the screen in hardware coordinates.

The *x* coordinates of a screen can range from 0 to 448 and are automatically rounded down to the nearest 16-pixel boundary. Only the positions between 112 and 448 are actually visible on your TV though, and you are strongly advised to avoid using an x coordinate below 112.

The *y* coordinates of your screen can range between 0 and 312. The visible region will largely depend on your TV or monitor, but you'll probably find that coordinates between 30 and 300 are satisfactory for the majority of systems.

At the time of writing, there appears to be a minor bug in the Amiga's HAM mode. These pictures cannot be displayed with a Y coordinate of exactly 256. So set your coordinates to intermediate values such as 255 or 257 instead. We're not sure if it's a hardware or software fault yet but it won't restrict you by any means.

w holds the width of your screen in pixels. If this is different from the original setting, only a part of your image will be shown, starting from the top left corner of the display. Like the x coordinates, the screen width will be rounded to the nearest 16 pixel boundary.

Similarly, *h* sets the apparent height of the screen. Changing this value will reduce the depth of your image.

Generally SCREEN OPEN will automatically select the display position for you using a standard setting in the AMOS configuration file. If a screen is larger than the display then AMOS sets the screen into overscan.

SCREEN DISPLAY provides you with a simple way of changing these values from the default. Any of the parameters *x,y,h, and w* may be omitted as appropriate. The unused values will be automatically assigned to the default settings, and should be separated by commas.

```
Screen Display 0,112,45,, : Rem Position the screen at 112,42
```

When you are positioning your screens, try to ensure that the screen starts at the left of the display and ends towards the right. This is essential if the Amiga's hardware is to interpret your screen correctly. In practice, you may need to experiment a little to get the precise effect you want. Fortunately, the worst that can happen is that you'll get a silly looking display. The Amiga won't crash if you make a mistake. Here are some guidelines to help you along:

- Only a single screen can be displayed on each horizontal line. However, you can safely place several screens on top of each other. All will be well, providing only one of the screens is visible.

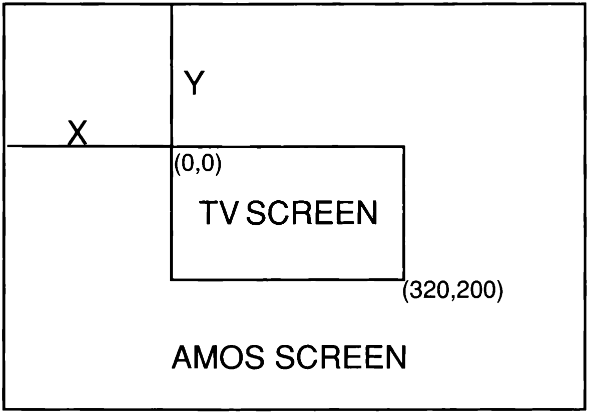
- There will always be a one pixel thick "dead zone" between each pair of screens. This is generated by the copper list and is completely unavoidable. The dead zone will be noticeable whenever you move a sprite between the screens. As an example, try moving the mouse pointer from the editor window to the menu line. You should see a small black line through your mouse pointer at the border between the two screens.

SCREEN OFFSET *(Hardware scrolling)*

SCREEN OFFSET n, x, y

The Amiga's display is not just limited to the visible dimensions of your TV screen. There's absolutely nothing stopping you from generating images which are much larger than the actual screen. It's obviously not possible to display such pictures in their entirety, but you can easily view a section of your image using the SCREEN OFFSET command. Look at the following diagram.

Using an extra large screen



As you can see, the selected area of the screen is displayed through a view port positioned at the coordinates x and y. You can move the view port through the whole screen by changing the values of these coordinates.This produces a smooth hardware scrolling effect which is perfect for many games.The size of your view port is taken from the dimensions you set in a previous call to the SCREEN DISPLAY command.

n is the number of the screen to be displayed. *x,y* measure the offset from the top left hand corner of the screen to the starting point for your display. *x and y* are specified in units of a single pixel, so there's nothing stopping you from generating some delightfully smooth scrolls.

You can also use negative offsets with this instruction, allowing you to display any part of the Amiga's memory on the screen. See **Example10.2** for a full demonstration of this command.

Screen control commands

SCREEN CLONE *(Clone a screen)*

SCREEN CLONE n

The SCREEN CLONE command assigns a second version of the current screen to screen number *n*. This *clone* uses exactly the same memory area as the original screen.

Normally, the cloned screen is displayed at the same place as its parent. However it can be manipulated separately using any of the normal screen operations such as SCREEN DISPLAY and SCREEN OFFSET.

Since there's only a **single** copy of the original screen data in memory, you can't access a clone with the SCREEN command. You'll get an *illegal screen parameter* error if you try. Another point to consider is that any colour flash sequences you've set up on the original screen will **not** be copied during the cloning operation. See **EXAMPLE 10.3**. Notice the use of the WAIT VBL command. This ensures that the clone is repositioned off-screen and keeps the movements running smoothly.

If you experiment with SCREEN CLONE, you'll quickly find that there's a real limit to the amount of movement you can perform without spoiling the effect completely. Even something as trivial as adding an extra calculation to your movement routine can often introduce an unacceptable delay into your animations.

The screen display can also be adjusted directly from the AMAL animation language. This is capable of animating large numbers of screens smoothly and easily. See **EXAMPLE 10.4** for a demonstration. The animations are now so fast you actually have to slow them down in order to see them!

DUAL PLAYFIELD *(Combine two screens into dual playfield)*

DUAL PLAYFIELD screen1, screen2

The Amiga's dual playfield mode allows you to display two complete screens simultaneously at the same x and y coordinates. It's almost as if you'd drawn each screen on cellophane and overlayed them on top of each other. Each screen can be manipulated totally independently. You can exploit this to produce a smooth parallax effect which is ideal for screen scrolling games such as Silkworm.

The two components of a dual playfield are treated just like any other AMOS screen and can be written to in the normal way. They can even be animated within AMAL or double buffered.

screen1 and *screen2* refer to screens which have been previously defined with the SCREEN OPEN command. Only certain screen combinations are acceptable. Both screens **must** use the same resolution, as it's illegal to use hires and lowres screens in the same playfield.

Here is a list of the possibilities

Screen 1	Screen 2	Notes
No of colours	No of colours	
2	2	
4	2	
4	4	
8	4	Lowres only
8	8	Lowres only

Although the colour ranges are predefined, the sizes of the two screens can be completely different. By creating a background screen which is larger than the foreground you can create a delightfully realistic parallax effect.

The colours of these screens are all taken from the palette of *screen1* with colour zero being treated as transparent.

Screen	Colour indexes (from screen 1)
1	0-7
2	8-15

When you are drawing to the second screen, AMOS Basic will automatically convert your colour index to the appropriate number before using it. So INK 2 will use colour nine from the first palette.

This conversion process does not apply to the assignment statements such as COLOUR or PALETTE. It's important to remember this when you are changing the colour settings, otherwise your new colours will not be reflected on the actual screen. Always make *screen1* the current screen before changing your colour assignments.

```
Screen 1 : Rem where screen 1 is the number of the first screen
```

There are a couple of impotant points which you must be aware of before setting up a dual playfield screen:

- The screen offsets for both screens must never be set to zero.
- If you set a dual playfield screen up and then want to position it with SCREEN OFFSET be sure to specify dual screen 1 not the second.

DUAL PLAYFIELD is an extremely powerful instruction. A full demonstration can be found in **EXAMPLE 10.5.**

DUAL PRIORITY *(Choose order of dual playfield screens)*

DUAL PRIORITY screen1,screen2

The first screen of a dual playfield is normally displayed directly over the second. The DUAL

PRIORITY command allows you to change this order around so that *screen2* appears in front of *screen* 1.

Warning! This instruction only changes the order of the display. It has **no** effect on the screen organization. The first screen in the dual playfield list should therefore still be used for all colour assignments and with SCREEN DISPLAY.

SCREEN *(Set current screen)*

SCREEN n

The SCREEN command allows you to direct all graphical and text operations to screen number *n.* If this screen is hidden or is positioned outside the display area, the command will have no visible effect. However the graphics will still be drawn in memory and they will be displayed when the screen is moved into view.

=SCREEN *(Get current screen number)*

s=SCREEN

Returns the number of the currently active screen. This is the screen which is used for all drawing operations but it is **not** necessarily visible.

SCREEN TO FRONT *(Moves screen to front of display)*

SCREEN TO FRONT [s]

This instruction moves screen *s* to the front of the TV display. If the parameter *s* is omitted, then the current screen will be used instead.

```
Screen Close 0 : Load Iff "AMOS_DATA:IFF/AMOSPIC.IFF",0
Load Iff "AMOS_DATA:IFF/Magic_Forest.IFF",1
Wait Key : Screen To Front 0
```

Note: If the AUTOVIEW system has been turned off, you'll need to call the VIEW command before the effect will be visible on the screen.

SCREEN TO BACK *(Move screen to back of the display)*

SCREEN TO BACK[n]

SCREEN TO BACK moves a screen to the background of your display. If there is another screen at the same coordinate this will now be displayed in front of the selected screen.

SCREEN HIDE *(Temporarily hide a screen)*

SCREEN HIDE [n]

Removes a selected screen from view completely. This screen can be redisplayed using a call to SCREEN SHOW. If *n* is omitted, this instruction will hide the current screen.

SCREEN SHOW *(Restore a screen)*

SCREEN SHOW n

Screen SHOW returns a screen onto the display after it has been hidden with the SCREEN HIDE command.

```
Load Iff "AMOS_DATA:IFF/AMOSPIC.IFF",1
Screen Hide : Wait Key : Screen Show
```

=SCREEN HEIGHT *(Return height of the current screen)*

h=SCREEN HEIGHT [n]

Returns the height of an AMOS screen. If you don't include a parameter with this instruction the height will be returned for the current screen.

```
Print Screen Height
```

=SCREEN WIDTH *(Return the width of the current screen)*

w=SCREEN WIDTH [n]

SCREEN WIDTH retrieves the width of either the current screen or screen number n.

```
Print Screen Width
```

=SCREEN COLOUR *(Return the number of colours)*

c=SCREEN COLOUR

The SCREEN COLOUR instruction returns the maximum number of colours in the currently active screen.

```
Print Screen Colour
```

=SCIN *(Returns screen number at a selelcted postion)*

s=SCIN(x,y)

Returns the number of the screen which is underneath the **hardware** coordinates *x,y.* If this screen does not exist, then *s* will be loaded with a negative value (null).

SCIN is normally used in conjunction with the X MOUSE and Y MOUSE functions to check whether the mouse cursor has entered a particular screen.Example:

```
Print Scin(X Mouse,Y Mouse)
```

Defining the screen colours

DEFAULT PALETTE *(Load screen with standard palette)*

DEFAULT PALETTE c1,c2,c3,,,c6,,,—> up to 32 colours

This command simplifies the process of opening many screens with the same palette. It defines a list of colours which will be used for all subsequent screens which you create with the SCREEN OPEN. instruction. As usual, the allowable colour values range from $000 to $FFF ($RGB). Also see GET SPRITE PALETTE.

GET PALETTE *(Set the palette from a screen)*

GET PALETTE n [,mask]

The GET PALETTE instruction copies the colours from screen *n* and loads them into the current screen. This can be very useful when you're moving information from one screen to another with SCREEN COPY, as it's usually vital that both the source and destination screens share the same colour settings.

The optional *mask* parameter allows you to load just a selection of the colours. See GET SPRITE PALETTE for full details of *mask.* Example:

```
Load iff "AMOS_DATA:Iff/AMOSPIC.IFF",0
Screen Open 1,Screen Width, Screen Height, Screen Colour, Lowres
Screen Copy 0,0,0,160,100 To 1,80,80
Centre "<Press a key to grab the palette>" : Wait Key
Get Palette 0
```

Clearing the screen

CLS *(Clear the screen)*

CLS erases all or part of the current screen. There are three possible formats of this command.

CLS

Clears the current screen by filling it with colour zero and clears any windows which may have been set up.

CLS col

Fills your screen with colour *col.*

CLS col,x1,y1 to x2,y2

Replaces the rectangular region at coordinates *x1,y1,x2,y2* with a block of colour *col. col* can take any value from 0 to the maximum number of available colours.

x1,y,x2,y2 hold the coordinates of the top left and bottom right corners of the area to be cleared by this command. Example:

```
Cls : Circle 100,98,98 : Cls 1,50,50 To 150,150
```

Manipulating the contents of a screen

SCREEN COPY *(Copy sections of the screen)*

SCREEN COPY scr1 TO scr2

SCREEN COPY scr1,x1,y1,x2,y2 TO scr2,x3,y3 [,mode]

SCREEN COPY makes it possible to copy large sections of a screen from one place to another at amazing speed.

scr1 holds the screen used as the source of your image. This can be either a standard screen number or the number of a logical or physical screen generated using the LOGIC and PHYSIC commands.

scr2 selects an optional destination screen into which this data will be copied. If it's omitted, the area will be copied into the current screen.

x1,y1 and *x2,y2* hold the dimensions of a rectangular source area, and *x3,y3* contain the coordinates of the destination. There are no limitations to these coordinates whatsoever. Any parts of your image which lie outside the current screen area will be automatically clipped as appropriate.

An example of SCREEN COPY can be found in **EXAMPLE 10.6** in the MANUAL folder.

The optional *mode* parameter chooses which of the 255 possible blitter modes will be used for your copying operation. These modes determine how your source and destination areas will be combined together on the screen. The mode is set using a bit-pattern in the following format:

Mode Bit	Source bit	Destination Bit
4	0	0
5	0	1
6	1	0
7	1	1

Note that the bottom four bits in the pattern are not used by this instruction and should always be set to zero.

Each bit in *mode* represents a single combination of bits in the source and destination areas. If a mode bit is set to one, then the associated bit on the screen will also be loaded with a one, otherwise the result will be zero.

In order to select the correct drawing mode for your application, you simply decide which combinations should result in a one and set the appropriate bits in the *mode* parameter accordingly.

Supposing you only wanted to set a bit on the screen if both the source and destination bits were the same. You would now look through the table for the points where your requirement was satisfied. This would produce the following value for *mode:*

%10010000

If you're not familiar with binary notation, you may find this command a little opaque. Rather than boring you silly with an explanation of binary we'll now provide you with a detailed list of the more common requirements along with the associated bit-maps.

Mode	Effect	Bit-pattern
REPLACE	Replaces the destination with a direct copy of the source image (default).	%11000000
INVERT	Replaces the destination image by a reversed copy of the source image.	%00110000
AND	Combines the source and destination with a logical AND operation.	%10000000
OR	OR's the source with the destination image.	%11100000
XOR	Combines the source and destination area with an Exclusive OR.	%01100000

Technically-minded users should note that SCREEN COPY combines the source and destination using blitter areas B and C and that blitter area A is not used by the system at all.

Scrolling the screen

DEF SCROLL *(Define a scrolling zone)*

DEF SCROLL n,x1,y1 to x2,y2,dx,dy

DEF SCROLL allows you to define up to 16 different scrolling zones. Each of these zones can be associated with a specific scrolling operation which is determined by the variables *dx* and *dy*.

n holds the number of the zone and can range from 1 to 16. *x1,y1* refer to the coordinates of the top left-hand corner of the area to be scrolled and *x2,y2* to the point diagonally opposite.

dx signifies the number of pixels the zone will be shifted to the right in each operation. Negative numbers indicate that the scrolling will be from right to left, and positive numbers from left to right.

Similarly, *dy* holds the number of points the zone will be advanced up or down during the scroll. In this case negative values of *dy* are used to indicate an upward movement and positive values a downward motion.

SCROLL *(Scroll the screen)*

SCROLL n

The SCROLL command scrolls the screen using the settings you have specified with the DEF SCROLL instruction. *n* refers to the number of the zone you wish to scroll.

```
                             AMOS PIC. IFF
Load Iff "AMOS_DATA:IFF/~~Frog_Leap.IFF~~",2
Def Scroll 1,0,0 to 320,200,1,0
Do
  Scroll 1
Loop
```

Larger examples can be found in **EXAMPLE 10.7 and EXAMPLE 10.8.** These load an image from the AMOS system disc and rotate it around on the screen. The variable *S* holds the number of points the picture will be moved during each SCROLL. The larger the value of S, the faster and jerkier the scrolling. Note the use of screen switching to improve the quality of the motion.

Screen switching

In order to produce the smooth movement effects found in a computer game, it's necessary to complete all the drawing operations within a time span of no more than a fiftieth of a second. This represents a real challenge for the fastest computer, and it's often impossible to achieve even on the Amiga. If the animation is complex, your graphics will therefore tend to flicker annoyingly as they are being drawn.

Fortunately, there's a solution at hand which has been successfully exploited in the vast majority of modern arcade games. This *screen switching* technique can easily generate flicker-free screen animation using just a fraction of the Amiga's computing power.

The basic idea is extremely simple. Instead of constructing your images on the actual screen, you perform all your drawing operations on a separate *logical screen.* which is completely invisible to the user. This is distinct from the *physical screen* which is currently being displayed on your TV. Once the graphics have been completed, you can then swap the logical and physical screens to produce a smooth transition between the two screen images. The old physical screen now becomes the new logical screen, and is used to construct the next picture in your sequence.

At first glance, this process looks pretty complicated, but it's all performed automatically by the AMOS Basic DOUBLE BUFFER command. This forces all drawing operations to be performed directly on the logical screen without affecting the current display. All you need to do within your program is to synchronise your drawing operations with the screen switches. This can be achieved with the help of the SCREEN SWAP instruction.

SCREEN SWAP *(Swap the logical and physical screens)*

SCREEN SWAP [n]

SCREEN SWAP swaps the physical and logical screens. This enables you to instantaneously switch the physical display between the two screens.

If you're using DOUBLE BUFFER, these screens will have been created for you already. However, you will need to switch off the automatic screen switching system with BOB UPDATE OFF, as otherwise the screens will be swapped 50 times a second, and will interfere with your own drawing operations. It's also necessary to kill the autoback feature with AUTOBACK OFF. This normally copies your graphical operations onto both physical and logical screens. It's useful when you wish to combine simple graphics with moving bobs, but it destroys the effect of your screen switching operations totally.

As an illustration of the power of this command, have a look at the programs **EXAMPLE 10.9** and **EXAMPLE 10.10.**

EXAMPLE 10.9 moves a triangle across the screen without using any form of screen switching. As the triangle proceeds, it generates an intense and annoying flicker. Now let's add a little screen switching to this program. We will draw the triangle on an invisible logical screen and flick it into view when it's been completely drawn. The new program can be found in **EXAMPLE 10.10.**

EXAMPLE 10.10 draws each new triangle on the screen without affecting the current display. The SCREEN SWAP instruction then swaps the logical and physical screens around, so the finished version of the triangle appears on the screen immediately, without a trace of flicker. The old triangle is now erased from the logical screen and redrawn at the next position. As the program runs, the triangle will smoothly progress from one side of the screen to the other.

Note that we've intentionally exaggerated the flicker in **EXAMPLE 10.9** to illustrate the screen switching technique. In practice it would be very easy to reduce this problem considerably even without the use of the SCREEN SWAP instruction.

=LOGBASE *(Return the address of part of the logical screen)*

address=LOGBASE(plane)

The LOGBASE function is aimed at expert programmers who wish to access the Amiga's screen memory directly.

plane refers to one of the six possible bit-planes which make up the current screen. After LOGBASE has been called, *address* will contain either the address of the required bit-plane, or zero if it doesn't exist.

=PHYBASE *(Return the address of the current screen)*

address=PHYSBASE(plane)

PHYSBASE returns the address in memory of bit-plane number *plane* for the current screen. If this plane does not exist, then a value of zero will be returned by this function. Example:

```
Loke Phybase(0),0 : Rem Pokes a thin line directly onto the screen
```

=PHYSIC *(Return identifier of the physical screen)*

=PHYSIC
=PHYSIC(s)

The PHYSIC function returns an identification number for the current physical screen. This number allows you to directly access the physical image which is being displayed by the double buffering system.

The result of this function can be substituted for the screen number in the ZOOM, APPEAR and SCREEN COPY commands.

s is the number of an AMOS screen. If it is omitted, then the present screen will be used instead. Do **not** confuse with the LOGBASE function.

=LOGIC *(Return identifier of the logical screen)*

=LOGIC
=LOGIC(s)

Returns an identification number of a logical screen. This can be used in conjunction with the SCREEN COPY, APPEAR and ZOOM commands to change your image off-screen, without affecting the current display.

Screen synchronisation

Like most home computers the AMIGA uses a memory-mapped display. This is a technical term for a concept you are almost certainly already familiar with. Put simply, a memory-mapped display is one which uses special hardware to convert an image stored in memory into a signal which can be displayed to your television screen. Whenever AMOS Basic accesses the screen it does so through the medium of this screen memory.

The screen display is updated by the hardware every 50th of a second. Once a screen has been drawn, the electron beam turns off and returns to the top left of the screen. This process is called the vertical blank period or VBL. At the same time, AMOS Basic performs a number of important tasks, such as moving the sprites and switching the physical screen address if it has changed. The actions of instructions such as ANIM or SCREEN SWAP will therefore only be fully completed when the screen is redrawn.

Since a 50th of a second is quite a long time for AMOS Basic, this can lead to a serious lack of coordination between your program and the screen, which is especially noticeable in tight program loops. The best way of avoiding this difficulty, is to wait until the screen has been updated before you execute the next Basic command.

WAIT VBL *(Wait for a vertical blank)*

The WAIT VBL instruction halts the AMIGA until the next vertical blank period. It is commonly used after either a PUT BOB instruction or a SCREEN SWAP.

Special effects

APPEAR *(Fade between two pictures)*

APPEAR source TO destination,effect [,pixels]

The APPEAR command enables you to produce fancy fades between the *source* and *destination* screens. *Source* and *destination* are simply the numbers of screens you've previously opened using SCREEN OPEN. You can also substitute the LOGIC and PHYSIC functions in these positions if required.

effect determines the type of fade which will be produced by this instruction. The size of this parameter can vary from 1 to the number of pixels in your current screen.

pixels specifies the number of points which are to be affected. Normally this value is set to the TOTAL screen area, but you can reduce it to fade only a part of the screen. All screens are drawn in strict order from the top of the screen to the bottom.

The appearance of your fades will naturally vary depending on the screen mode you are using. A program is provided in **EXAMPLE 10.11** to allow you to experiment with the various possibilities.

FADE *(Blend one or more colours to new colour values)*

FADE speed [,colour list]
FADE speed TO screen [,mask]

The FADE command allows you to smoothly change the entire palette from one set of colours to another. This can be used to generate professional-looking fade effects for your loading screens.

The standard version of the instruction takes the current palette, and slowly dissolves the screen colours to zero. Each colour value is successively reduced by one until they reach zero. Example:

```
Fade 15:Wait 225
```

speed is the number of vertical blank periods that must occur before the next colour change is performed.

Since the fading effects are executed using interrupts, it's best to wait until the operation has completely finished before proceeding to the next Basic instruction. The time taken for the fade WAIT can be calculated by the formula:

wait value = fade speed * 15

FADE can be extended to generate a new palette directly from a list of colour values.

```
Fade 15,$100,$200,$200,$300
```

Any number of colours can be specified in this instruction, up to the maximum allowed in the current graphics mode. Like most AMOS commands, it's possible to omit selected parameters completely. These colours will be totally unaffected by the FADE command.

```
Fade 15,,$100,$800,$F00
```

The most powerful form of FADE smoothly transforms the colours from the current screen into a palette taken from an existing screen.

FADE speed TO s [,mask]

The present colours are slowly converted into the palette of screen *s*. It's also possible to load the palette from the sprite bank using the same technique. Simply use a negative value for the screen number *s.*

mask is a bit-pattern which specifies which colours should be loaded. Each colour is associated with a single bit in this pattern numbered from 0 to15. If a bit is set to 1, then the relevant colour will be changed. See **EXAMPLE 10.12.**

FLASH *(Set flashing colour sequence)*

This command gives you the ability to periodically change the colour assigned to any colour index. It does this with an interrupt similar to that used by the sprite and the music instructions. The format of the flash instruction is:

```
FLASH index,"(colour, delay)(colour, delay)(colour,delay)..."
```

Index is the number of the colour which is to be animated. *Delay* is set in units of a 50th of a second.

Colour is stored in the standard RGB format (See COLOUR for more details).The action of FLASH is to take each new colour from the list in turn, and then load it into the index for a length of time specified by the *delay.* When the end of this list is reached, the entire sequence of colours is repeated from the start. Note that you are only allowed to use a maximum of 16 colour changes in any one FLASH instruction. Here is a small example:

```
Flash 1,"(007,10)(000,10)"
```

This alternates colour number 1 between blue and black every 10/50 (1/5th) of a second. Now for something more complex:

```
Flash 0,"(111,2)(333,2)(555,2)(777,2)(555,4)(333,4)"
```

If this gives you a headache, you will be glad to learn that you can turn the flashing off using the instruction:

```
FLASH OFF
```

Also note that on start-up, colour number 3 is automatically assigned a flash sequence for use by the cursor. It's a good idea to turn this off before loading any pictures from the disc.

SHIFT UP *(Colour rotation)*

```
SHIFT UP delay,first,last,flag
```

The SHIFT UP command rotates the values held in the colour registers from *first* to *last.* The *first* colour in the list is copied into the second, the second into the third, and so on, until the *last* colour in the series is reached.

Each AMOS screen can have its own unique set of colour animations. Colour shifts can be used to create amazing hyperspace sequences similar to those found in Captain Blood and Elite. Since these animations are performed using interrupts, they can be executed while your program is running, without affecting it in the slightest.

delay is the time interval between each stage of the rotation, measured in 50ths of a second.

flag controls the type of rotation. If its set to one, the last colour index in the list will be copied into the first, and the first to the last. So the colours will rotate continuously on the screen. When *flag* is set to zero, the contents of the *first* and *last* indexes will be discarded, and the region between *first* and *last* will gradually be replaced by a copy of the first colour in the list. For example:

```
Shift Up 100,1,15,1

Shift Up 10,1,15,0
```

These colour shifts can be turned off at any time using the SHIFT OFF command.

SHIFT DOWN *(Rotate a list of colour values down)*

SHIFT DOWN delay,first,last,flag

SHIFT DOWN is similar to the previous SHIFT UP command, except that it rotates the colours in the opposite direction. So the second colour will be copied into the first, the third colour into the second, and so on.

first and *last* input a list of colour indices to be rotated. *delay* species an interval between each colour shift in units of a 50th of a second.

flag sets the type of rotation. A value of one results in a continuous colour cycle, and a zero shifts the colours without saving the original contents of *first* and *last.* After a complete cycle, all the colours between *first* and *last* will contain a copy of the colour held in *last.* See also FLASH, PALETTE and COLOUR.

SHIFT OFF *(Stops all colour cycles for the current screen)*

SHIFT OFF

SHIFT OFF immediately terminates all colour rotations produced by the SHIFT UP or SHIFT DOWN instructions.

SET RAINBOW *(Define a rainbow effect)*

SET RAINBOW n,colour,length,r$,g$,b$

SET RAINBOW defines an attractive rainbow effect which can be subsequently displayed using the RAINBOW command. It works by changing the shade of a colour according to a series of simple rules.

n is the number of your rainbow. Possible values range from 0 to 3. *colour* is a colour index which will be changed by the instruction. This colour can be assigned a different value for each horizontal screen line (or *scan line*). Note that only colours from 0 to 15 can be manipulated using this system.

length sets the size of the table used to store your colours. There's one entry in this table for each colour value on the screen. The size of this table can range from 16 to 65500. If *length* is less than the physical height of your rainbow, then the colour pattern will be repeated several times on the screen.

The *r$,g$,b$* command strings, progressively change the intensities of the red, green and blue components of your final colour. These values are loaded into a special colour table. Each colour in the table determines the appearance of a single horizontal scan line on the screen.

At the start of the rainbow, all the components in your colour are initially loaded with a value of zero. This will be changed according to the information held in the colour table.

Any command string may be omitted if required, but you'll still have to include the quotes and the commas in their expected positions.

Each string can contain a whole list of commands. These will be cycled continually to produce the final rainbow pattern.The format is:

(n,step,count)

n sets the number of lines to be assigned to a specific colour value in the rainbow. Increasing this number will change the height of each individual rainbow line.

step holds a number to be added to the component. This number will be used to generate the colour of the succeeding line on the screen. A positive *step* will increase the intensity of colour component, and a negative value will reduce it.

Whenever a particular component exceeds the maximum of 15, a new value will be calculated from the formula:

new component=old component Mod 15

count is the number of times the current operation is to be repeated. The best way to demonstrate this command is with an example. Type in the following lines into your computer.

```
Set Rainbow 0,1,64,”(8,2,8)”,””,””
Rainbow 0,56,1,255 : Rem Displays rainbow
Wait key
```

This creates a new rainbow with number zero using colour index one. As you can see, SET RAINBOW only defines your rainbow. In order to display it on the screen you need to make use of the RAINBOW command (see below).

The rainbow effect first loads your colour with a value of zero. Every four scan-lines, the red component will be automatically incremented by two. So the contents of colour zero will progressively change from $000 to $E00. When the component exceeds the maximum of 15, its remainder will be calculated, and the colour will be returned to its starting point (zero). The pattern will now be repeated down the screen.

By defining a separate pattern for each of the red, green and blue components of your colour, you can easily generate some startling patterns on the screen. Since each rainbow only uses a single colour index, there's nothing stopping you from creating the same effects using just two colour screens. These are ideal for the backgrounds of an arcade game, as they consume very little memory. Example:

```
Screen Open 0,320,256,2,Lowres
Set Rainbow 0,1,128,”(8,1,8)”,”(8,1,8)”,””
Rainbow 0,1,30,128
```

```
Colour 1,0 : Curs Off : Cls 1 : Flash Off
Locate 0,2 : Centre "Amos Basic" : Wait Key.
```

For a further demonstration of the superb effects that can be achieved with this instruction, load up **EXAMPLE 10.13**.

Rainbows can also be animated using a powerful interrupt system. See the section on AMAL for more details.

RAINBOW *(Create a rainbow effect)*

RAINBOW n,base,y,h

RAINBOW displays rainbow number *n* on the screen. If AUTOVIEW is set to OFF, the rainbow will only appear when you next call the VIEW command.

base is an offset in the first colour in the table you created with SET RAINBOW. Changing this value will cycle the rainbow on the screen.

y holds the vertical position of the rainbow in hardware coordinates. The minimum value for this coordinate is 40. If you attempt to use a coordinate below this point, the rainbow will be displayed from line 40 onwards.

h sets the height of your rainbow in scan lines.

Rainbows are totally compatible with the AMOS system including bobs and sprites. However, don't attempt to rainbow a colour which is currently being changed using the FLASH or SHIFT instructions, as this will lead to unpredictable screen effects.

Note that only a single rainbow effect can be displayed on a particular scan line, even if they use different colours on the screen.

Normally the rainbow with the highest screen position will be displayed first. But if several rainbows start from the same scan line, then the rainbow with the lowest identification number will be drawn in front of the others.

=RAIN *(Change the colour of an individual rainbow line)*

RAIN(n,line)=c
c=RAIN(n,line)

This is the most powerful of all the rainbow creation commands, as it allows to change the colour of an individual rainbow line to any value you like.

n is the number of the rainbow you wish to access. *line* is the individual scan-line to be changed. Example:

```
Curs Off:Centre "An AMOS Rainbow!"
Set Rainbow 1,1,4097,"","","" : Rem set up rainbow with dummy values
For Y=0 To 4095
  Rain(1,Y)=Y : Rem Load rainbow with a colour value from 0-4095
Next Y
For C=0 To 4095-255
  Rainbow 1,C,40,255 : Rem display rainbow 255 lines long starting at 40
Next C
Wait Key
```

This smoothly scrolls the entire palette through colour number one.

ZOOM *(Magnify a section of the screen)*

ZOOM source,x1,y1,x2,y2 TO dest, x3,y3,x4,y4

ZOOM is a simple instruction which allows you to change the size of any rectangular region of the screen.

Source is the number of a screen from which your picture will be taken. You can also use the LOGIC function to grab your image from the appropriate logical screen. The rectangular area to be affected by this instruction is entered using the coordinates *x1,y1,x2,y2. x1,y1* holds the position of the top left hand corner of this region and *x2,y2* sets the coordinates of the corner diagonally opposite. *dest* holds the destination screen for your image. Like the source, it can be either a screen number, or a logical screen specified using LOGIC.

The dimensions of this screen are taken from the coordinates *x3,y3* and *x4,y4.* These hold the dimensions of the rectangle into which the screen segment will be compressed.

The effect of this instruction depends on the relative sizes of the source and destination rectangles. The source image is automatically resized to fit exactly into the destination rectangle. So the same instruction can be used to reduce or enlarge your images as required. Here's an example:

```
F$=Fsel$("*.*","","Load screen") : If F$="" then Direct
Load Iff F$,0 : Screen Open 1,320,256,Screen Colours,Lowres
Flash off : Get Palette(0)
Screen Display 1,,,,256 : View : Limit Mouse
Repeat
Zoom 0,0,70,320,175 To 1,0,0,X Screen (X Mouse)+1,Y Screen (Y MOUSE)+1
Until Mouse Key
```

A further demonstration can be found in **EXAMPLE 10.14.**

Changing the copper list

The Amiga's coprocessor (copper) provides total control over the appearance of every line on your screen. This copper is a separate processor with its own internal memory and unique set of instructions. By programming the copper it's possible to freely generate a massive variety of different screen effects. Normally the copper is managed automatically by the AMOS system. Each of the available copper effects can be performed directly from within AMOS Basic without the need to indulge in complicated machine-level programming. In practice these instructions will be more than sufficient for the vast majority of applications.

Obviously, no one can think of everything though. Expert programmers may wish to access the copper directly to create their own special screen modes.

Be warned: The copper list is notoriously difficult to program, and if you don't know precisely what you are doing, you'll almost certainly crash your Amiga. Before embarking on your copper experiments for the first time, you are therefore advised to read one of the many reference books on the subject. A good explanation can be found the *Amiga System Programmers Guide* from Abacus.

COPPER OFF *(Turn of the standard copper list)*

COPPER OFF

This freezes the current AMOS copper list and turns off the screen display completely. You can now create your own display using a series of COP MOVE and COP WAIT instructions.

As a default, all user-defined copper lists are limited to a maximum of 12k. On average, each copper instruction takes up two bytes. So there's space for around six thousand instructions. This may be increased if required, using a special option from the CONFIG utility.

Note that all copper instructions are written to a separate logical list which is not displayed on the screen. This stops your program corrupting the display while the copper list is being created. To activate your new screen, you'll need to swap the *physical* and *logical* lists around with the COP SWAP command.

It's also important to generate your copper lists in strict order, starting from the top left of your screen and progressing downward to the bottom right. See **EXAMPLE 10.15**. in the MANUAL folder for a demonstration.

COPPER ON *(Restart the copper list)*

COPPER ON

COPPER ON restarts the AMOS copper list calculations and displays the current AMOS screens. Providing you haven't drawn anything since the COPPER OFF instruction, the screen will be restored to precisely its original state.

COP MOVE *(Write a MOVE instruction into the logical copper list)*

COP MOVE addr,value

Generates a MOVE instruction in the logical copper list.

addr is an address of a 16 bit register to be changed. This must lie within the normal copper DATA ZONE ($7F-$1bE). *value* is a word-sized integer to be loaded into the requested register.

COP MOVEL *(Write a long MOVE instruction into copper list)*

COP MOVEL addr,value

This is identical to the standard COP MOVE command, except that *addr* now refers to a 32-bit copper register. *value* contains a long word integer.

COP WAIT *(Copper WAIT instruction)*

COPY WAIT x,y[, x mask, y mask]

COP WAIT writes a WAIT instruction into your copper list. The copper waits until the hardware coordinates x,y have been reached and returns control to the main processor.

Note that line 255 is automatically managed by AMOS. So you don't have to worry about it at all.

x mask and *y mask* are bit maps which allow you to wait until just a certain combination of bits in the screen coordinates have been set. As a default both masks are automatically assigned to $1FF.

COP RESET *(Reset copper list pointer)*

COP RESET

COP RESET restores the address used by the next copper instruction to the start of the copper list.

=COP LOGIC *(Address of copper list)*

addr=COP LOGIC

The COP LOGIC function returns the absolute address in memory of the logical copper list. This allows you to poke your COPPER instructions directly into the buffer, possibly using assembly language.

Hints and tips

- Before creating a screen with a user-defined copper list, you'll first need to allocate some memory for the appropriate bit-maps. Although you can use RESERVE for this purpose, it's much easier to define a dummy screen with the SCREEN OPEN command instead. The copper registers can be loaded with the addresses of the required bit-maps using the LOGBASE function.

  You'll now be able to access your screen using all the standard AMOS drawing features. In order to reserve the correct amount of memory, set the number of colours to the **maximum** used in the new screen. This may be a little wasteful, but simplifies things enormously.

- It's perfectly acceptable to combine user-defined screens with AMOS bobs. If you're using double buffering though, you'll have to define a separate copper list for both the logical and physical screens. This may be achieved using the following procedure;

  1 Define your copper list for the first screen
  2 Swap the logical and physical copper lists with COP SWAP
  3 Swap the physical and logical screens with SCREEN SWAP
  4 Define your copper list for the second screen

  This will ensure that your bobs will be updated correctly on your new screens. All the normal AMOS commands can be used including AMAL.

11: Hardware sprites

One of the biggest attractions of the Commodore Amiga is its ability to produce high quality games which rival those found on genuine arcade machines. This can be amply demonstrated by terrific programs such as Battle Squadron and Eliminator.

Now, for the first time, all these amazing features are at your fingertips! AMOS Basic provides you with complete control over the Amiga's hardware and software sprites. These sprites can be effortlessly manoeuvred with the built-in AMAL animation language, so you don't have to be a machine code wizard in order to create your own stunning arcade games.

Hardware sprites are separate images which can be automatically overlayed on the Amiga's screen. The classic example of a hardware sprite is the mouse pointer. This is completely independent of the screen, and works equally well in all the Amiga's graphics modes.

Since sprites don't interfere with the screen background, they are perfect for the moving objects required by an arcade game. Not only are they blindingly fast, but they also take up very little memory. So when you're writing an arcade game, hardware sprites should always be at the top of your list.

Each sprite is 16 pixels wide and up to 255 pixels high. The Amiga's hardware supports a maximum of eight three-colour sprites or four fifteen-colour sprites. Colour number zero is transparent – that's the reason for the odd colour ranges.

At first glance, these features don't seem particularly impressive. But there are a couple of useful tricks which can increase both the numbers and sizes of these sprites beyond recognition.

One solution is to take each hardware sprite and split it into a number of horizontal segments. These segments can be independently positioned, allowing you to apparently display dozens of sprites on the screen at once. Similarly, the width restriction can be exceeded by building an object out of several individual sprites. Using this technique it's easy to generate objects up to 128 pixels wide.

Until recently the only way to exploit these techniques was to delve into the mysterious world of 68000 assembler language. So you'll be delighted to discover that AMOS Basic manages the entire process automatically! Once you've designed your sprites with the AMOS sprite editor, you can effortlessly manipulate them with just a single Basic instruction.

The sprite commands

Remember to have a sprite bank loaded into memory when trying out the various commands in this chapter. We advise you use the file SPRITES.ABK from the AMOS data disc.

SPRITE *(Display a hardware sprite on the screen)*

SPRITE n,x,y,i

The SPRITE command displays a hardware sprite on the screen at the coordinates *x,y* using image number *i.*

N is the identification number of the sprite and can range from 0 to 63. Each sprite can be associated with a separate image from the sprite bank, so the same image can be used for several sprites.

x and *y* hold the position of the sprite using special hardware coordinates. All measurements are taken from the *hot spot* of your images. This serves as a sort of 'handle' on the sprite and is used as a reference point for the coordinates. Normally the hot spot is set to the top left hand corner of an image. However it can be changed within your program using the HOT SPOT command.

Hardware coordinates are independent of the screen mode and effectively start from (-129,-45) on the default screen. AMOS provides you with several built-in functions for conversions between hardware coordinates and the easier to use screen coordinates. See the X HARD, Y HARD, X SCREEN and Y SCREEN functions for more details.

I is the number of a particular image stored in the sprite bank. This bank can be created using the AMOS sprite editor, and is automatically saved along with your Basic program. It can also be loaded directly with the LOAD instruction. In addition you can use the GET SPRITE command to grab an image straight off the current screen.

Any of the parameters *x,y,* and *i* may be optionally omitted, but the appropriate commas **must** be included. For example:

```
Load "AMOS_DATA:Sprites/Octopus.abk"
Sprite 8,200,100,1
Sprite 8,,150,1
Sprite 8,300,,
```

For a demonstration of sprites in action, load **EXAMPLE 11.1** from the MANUAL folder on the AMOS data disc.

Computed sprites

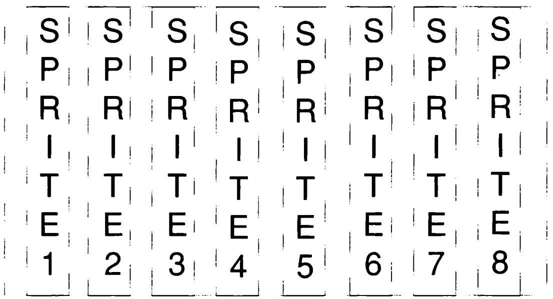
Although the Amiga only provides you with eight actual sprites, it's possible to use them to display up to 64 different objects on the screen at once. These objects are known as *computed sprites* and are managed entirely by AMOS Basic. Computed sprites can be assigned by calling the SPRITE command with a number greater than seven. For example:

```
Load "AMOS_DATA:Sprites/Octopus.abk"
Sprite 8,200,100,1
```

The size of a computed sprite is taken directly from the image data, and can vary between 16 and 128 pixels wide, and from 1 to 255 pixels high.

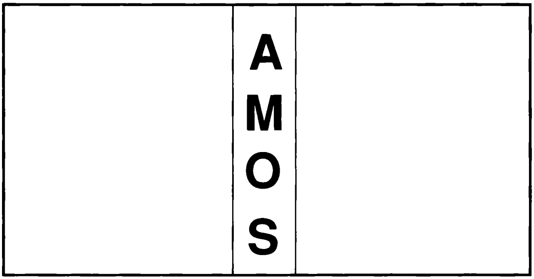
Before you can make full use of these sprites you need to understand some of the principles behind them. Each hardware sprite consists of a thin narrow strip 16 pixels wide and up to 256 pixels deep. Depending on the number of colours, you can have either eight or four of these strips on the screen at a time.

THE HARDWARE SPRITES



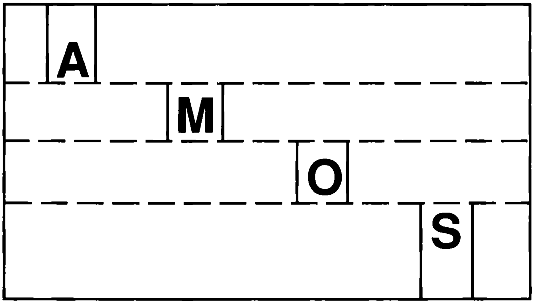
It should be obvious that most of the area inside these sprites is effectively wasted. That's because few programs need sprites which are taller than about 40 or 64 pixels. However there is a simple trick which enables us to borrow this space to generate dozens of extra objects on the screen. Look at the sprite below which contains the letters A, M, O and S.

A single hardware sprite



This sprite can be split into four horizontal segments each enclosing a single letter. The Amiga's hardware allows each section to be freely positioned anywhere on the current line, making a total of four *computed* sprites. Here's a diagram which illustrates this process.

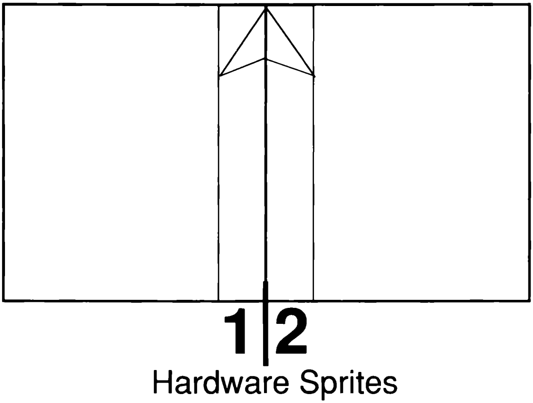
Splitting a hardware sprite into computed sprites



As you can see, a computed sprite is really just a small part of a hardware sprite displayed at a different horizontal screen position. Notice the line between each object. This is an unavoidable side effect of the repositioning process, and is generated by the Amiga's hardware.

Due to the way computed sprites are produced, there are a couple of restrictions to their use. Firstly, you can't have more than eight computed sprites on a single line. In practice the system is complicated by the need to produce sprites which are larger than the 16 pixel maximum. AMOS generates these objects by automatically positioning several computed sprites side by side. This can be seen from the diagram below:

Combining two sprites together



The maximum of eight hardware sprites therefore imposes a strict limit to the number of such objects you can display on a horizontal line. The total width of the objects must not exceed:

16*8=128 pixels for three-colour sprites

16*4=64 pixels for fifteen-colour sprites

If you attempt to ignore this limitation, you won't get an error message, but your computed sprite will not be displayed on the screen. So it's vital to ensure that the above restriction is never broken.This can be achieved using the following procedure:

Add together the widths of all your computed sprites, multiplying the dimensions of any fifteen-colour sprites by two. If the total is greater than 128, you'll need to space your sprites on the screen so that their combined width lies below this value. Take particular care if you are animating your sprites with AMAL as certain combinations will only come to light after you've experimented with the sequence for some time. These problems will be manifested by the random disappearance of one or more sprites on the screen.

If the worst comes to the worst, you'll need to substitute some of your larger sprites with Blitter Objects. This will increase the overall size of your program significantly, but it should have a negligible effect on the final quality of your game.

These restrictions are not confined to AMOS Basic of course. They apply equally well to all games on the Amiga, even if they're written entirely in machine code! So there's nothing stopping you from producing your own Xenon II clone using exactly the same techniques.

Note that, normally, hardware sprite number zero is allocated to the mouse cursor. You can release this sprite with a simple call to the HIDE command. See **EXAMPLE 11.2**.

Creating an individual hardware sprite

The only real problem with computed sprites is that you never know precisely which hardware sprite is going to be used in a particular object. Normally the hardware sprites used in an object will change whenever the object is moved. Occasionally this can be inconvenient, especially when you are animating objects such as missiles which need to remain visible in a wide range of possible sprite combinations.

In these circumstances it's useful to be able to allocate a hardware sprite directly. Individual hardware sprites can be assigned using the SPRITE instruction with an identification number between 0 and 7. Example:

```
Sprite 1,100,100,2
```

This loads hardware sprite number 1 with image number 2. *N* now corresponds to the number of a single hardware sprite, and can range between 0 and 7. If your image is larger than sixteen pixels wide, AMOS will automatically grab the required sprites in consecutive order, starting from the sprite you have chosen. For example:

```
Sprite 2,200,100,1
```

Supposing image number 1 contained a 32-bit image with three colours. This command would allocate hardware sprites 2 and 3 to the image. Nothing would happen if you were now to attempt to display hardware sprite 3 with a command like SPRITE 3,150,100,1 because this sprite has already been used. You would only have access to sprites 0,1,4,5,6 and 7, and the maximum numbers and sizes of your computed sprites would be reduced accordingly.

Each 15-colour sprite is implemented using a pair of two three-colour sprites. However, it's not possible to combine any two sprites in this way. Only the combinations 0/1,2/3,4/5,6/7 are allowed. One side effect of this, is that you should always assign your hardware sprites using even sprite numbers. Otherwise, AMOS will start your sprite from the next group of two, effectively wasting the first sprite.

Also note that if you try to create a large fifteen-colour sprite with this system, you could easily use up all the available sprites in a single object.

Warning! If you are writing a screen scrolling game, you may encounter problems using sprites in conjunction with the SCREEN OFFSET and SCREEN DISPLAY commands. These generate a DMA clash between the sprite system and the screen bit-maps, and can occasionally lead to unwanted screen effects.

This problem is only relevant if you are using hardware sprites 6/7. When the screen is shifted to the left with SCREEN OFFSET, the amount of time for your sprite updates is reduced, as the screen DMA has priority over the sprite system. Unfortunately, there isn't enough processing time to draw sprites 6/7, and they will be therefore be corrupted on your display.

To clear up this problem, create sprites 6/7 as individual hardware sprites and position them off the screen using negative coordinates. This will stop AMOS Basic from using them in your computed sprites. Providing sprites 6/7 are never displayed on the screen during your scrolling operations, all will be well.

The sprite palette

The colours required by a hardware sprite are stored in the colour registers 16 to 31. Providing your current screen mode doesn't make use of these registers, the sprite colours will be completely separate from your screen colours. Interestingly enough, this is also the case for the 4096-colour Ham mode. So there's nothing stopping you from producing some mind-blowing Ham games with this system!

However you will encounter real problems when using 32 or 64 colour screens in conjunction with three colour sprites. This is because the colours used by these sprites are grouped together in the following way:

Hardware sprites	Colour registers
0/1	17/18/19
2/3	21/22/23
4/5	25/26/27
6/7	29/30/31

Colour registers 16,20,24 and 28 are treated as transparent.

The difficulty arises due to the way AMOS generates computed sprites. The hardware sprites used to produce these objects vary during the course of a game, so it's vital to ensure that the three colours used by each individual sprite are set to exactly the same values, otherwise the colours of your computed sprites will change unpredictably. Here's a small AMOS procedure which will perform the entire process for you automatically.

```
Procedure INIT_SPRITES
   Get Sprite Palette
   For S=0 To 3
      For C=0 To 2
         Colour S*4+C+17,Colour(C)
      Next C
   Next S
Endproc
```

The above restriction does not, of course, apply to fifteen-colour sprites. If you want to make the most of the Extra Half Bright or 32-colour modes, you may find it easier to avoid using four-colour sprites altogether.

GET SPRITE PALETTE *(Grab sprite colours into screen)*

GET SPRITE PALETTE [mask]

This loads the entire colour palette used for your sprite images into the current screen. The optional *mask* allows you to load just a selection of the colours from the sprite palette. Each of the 32 colours is represented by a single bit in the mask, numbered from right to left. The right-most bit represents the status of colour zero, the next bit colour 1, and so on. To load a colour, simply set the appropriate bit to 1. If, for instance, you wanted to copy just the first four colours, you would set the bit pattern to:

```
Get Sprite Palette %0000000000001111
```

Incidentally, since bobs use the same sprite bank as sprites, this command can also be used to load the colours of a bob.

Controlling sprites

SET SPRITE BUFFER *(Set height of the hardware sprites)*

SET SPRITE BUFFER n

This sets the work area in which AMOS creates the images of the hardware sprites. Acceptable values for *n* range from 16 to 256. To set the correct value for *n*, simply examine the sprites in the sprite editor and work out which is the largest sprite length wise. Any sprite that is larger than *n* will simply be truncated at the appropriate cut off point.

SET SPRITE BUFFER is supplied for your use so that you can claim back any redundant memory that your game or application simply does not use.

The amount of memory consumed by the sprite buffer can be calculated using the formula:

Memory = N*4*8*3 = N*96

So the minimum buffer size is 1536 bytes and the maximum is 24k. Note: This command erases all current sprite assignments and resets the mouse cursor to its original state.

SPRITE OFF *(Remove one or more sprites from the screen)*

SPRITE OFF [n]

The SPRITE OFF command removes one or more sprites from the screen. All current sprite movements are aborted. In order to restart them, you'll need to completely reinitialize your movement pattern.

SPRITE OFF	Removes all the sprites from display
SPRITE OFF n	Only deactivates sprite *n*

Note that your sprites are automatically deactivated whenever you call up the AMOS Basic editor. They will be autmatically returned to their original positions the next time you enter direct mode.

SPRITE UPDATE *(Control sprite movements)*

SPRITE UPDATE [ON/OFF]

The SPRITE UPDATE command provides you with total control of the movements of your sprites. Normally, whenever you move a sprite, its position is updated automatically during the next vertical blank period (see WAIT VBL). But if you are moving a lot of sprites using the SPRITE command, the updates will occur before all the sprites have been moved. This may result in a noticeable jump in your movement patterns. In these circumstances, you can turn off the automatic updating system with the SPRITE UPDATE OFF command.

Once your sprites have been successfully moved, you can then slide them smoothly into place with a call to SPRITE UPDATE. This will reposition any sprites which have moved since your last update.

See UPDATE EVERY, UPDATE and BOB UPDATE.

=X SPRITE *(Get X coordinate of a sprite)*

x = X SPRITE(n)

Returns the current X coordinate of sprite *n*, measured using the hardware system. This command allows you to quickly check whether a sprite has passed off the edge of the Amiga's screen.

=Y SPRITE *(Get Y coordinate of a sprite)*

y = YSPRITE(n)

Y SPRITE returns a sprite's vertical position. As usual, *n* refers to the number of the sprite and can range from 0 to 63. Remember, all sprite positions are measured in *hardware* coordinates. See **EXAMPLE 11.3**.

GET SPRITE *(Load a section of the screen into the sprite bank)*

GET SPRITE [s,] i,x1,y1 To x2,y2

This instruction enables you to grab images directly off the screen and turn them into sprites. The coordinates *x1,y1* and *x2,y2* define a rectangular area to be captured into the sprite bank. Normally all images are taken from the current screen. However it's also possible to grab the image from a specific screen using the optional screen number *s*.

Note: There are no limitations to the region that may be grabbed in this way. Providing your coordinates lie inside the existing screen borders, everything will be fine.

I denotes the number of the new image. If there is no existing sprite with this number, a new image will be created automatically. AMOS will also take the trouble of reserving the sprite bank if it hasn't been previously defined. See **EXAMPLE 11.4**.

There's also an equivalent GET BOB instruction which is identical to GET SPRITE in every respect. Since the sprite bank is shared by both bobs and sprites, the images are in exactly the same format. So it's perfectly acceptable to use both instructions in conjunction with either bobs or sprites. Try changing the sprite instruction in the previous example to something like:

```
Bob 1,0,0,1
```

Conversion functions

=X SCREEN *(Convert hardware coordinates into screen coords)*
=Y SCREEN

X=X SCREEN([n,] xcoord)
Y=Y SCREEN([n,] ycoord)

Transforms a hardware coordinate into a screen coordinate relative to the current screen. If the hardware coordinates lie outside the screen then both functions will return relative offsets from the screens boundaries. Type the following from direct mode:

```
Print X Screen(130)
```

The result will be -2. This is because the x screen coordinate 0 is equal to hardware coordinate 128 and thus the conversion of 130 to a screen coordinate results in a position two pixels to the left of the screen.

If the optional screen number is included then the coordinates will be returned relative to screen *n.*

=X HARD *(Convert screen coordinates into hardware coordinates)*
=Y HARD

X=XHARD ([n,] xcoord)

These functions convert a screen coordinate into a hardware coordinate. There are four separate conversion functions, the above syntax converts *xcoord* from a coordinate relative to the current screen to a hardware coordinate.

Y=Y HARD([n,] ycoord)

Transforms a Y coordinate relative to the current screen into a hardware coordinate. As before *n* specifies a screen number for use with the functions. All coordinates will now be returned relative to this screen.

=I SPRITE *(Return current image of a sprite)*

Image=I Sprite(n)

This function returns the current image number being used by sprite n. A value of zero will be reported if the sprite is not displayed.

12: Blitter objects

While hardware sprites are certainly powerful, they do suffer from a couple of annoying restrictions. Not only is there a maximum of eight sprites per horizontal line but each sprite is also limited to just fifteen colours.

The solution is to make use of the Amiga's infamous Blitter chip. This is capable of copying images to the screen at rates approaching a million pixels per second! With the help of the blitter it's possible to create what are known as *bobs* (or Blitter objects).

Bobs, like sprites, can be moved around completely independently of the screen without destroying any existing graphics. But unlike sprites, bobs are stored as part of the current screen, so you can create them in any graphics mode you wish. This allows you to generate blitter objects with up to 64 colours. Furthermore the only limit to the number of bobs you can display is dictated by the available memory.

Of course all this power doesn't come without a price. In practice bobs are slightly slower than sprites and they consume considerably more memory. Therefore there's a trade-off between the speed of sprites, and the flexibility of bobs. Fortunately there's nothing stopping you from using both bobs and sprites in the same program. This is the approach which is adopted in the majority of commercial games. Bobs are used for larger objects like spaceships, while hardware sprites are often reserved for small fast–moving objects such as missiles.

BOB *(Draw a blitter object on the current screen)*

BOB n,x,y,i

The BOB command creates bob *n* at coordinates *x,y* using image number *i.*

n is the identification number of the bob. Permissible values normally range from 0 to 63, but the number of bobs may be increased using an option from the AMOS configuration program. Providing you've enough memory, you can set this limit to any number you wish.

x and *y* specify the position of the bob using standard *screen* coordinates. These coordinates are not the same as the hardware coordinates used by the equivalent SPRITE command. Like sprites, each bob is controlled through a *hot spot.* This may be changed at any time with the HOT SPOT command.

i refers to an image which is to be assigned to the bob from the sprite bank. The format of this image is identical to that used by the sprites, so you can use the same images for either sprites or bobs.

After you've created a bob, you can independently change either its position or its appearance by omitting one or more parameters from this instruction. Any of the numbers *x, y* or *image* may be left out, with the missing parameters retaining their original values. This is particularly useful if you are animating your bob with AMAL, as it allows you to move your object anywhere you like, without disturbing your existing animation sequence. However you must always include the commas in their original order, otherwise you'll be presented with a syntax error. For example:

```
Load "AMOS_DATA:Sprites/Octopus.abk"
Flash Off : Get Sprite Palette
Channel 1 To Bob 1
Bob 1,0,100,1
```

```
Amal 1,"Anim 0,(1,4)(2,4)(3,4)(4,4)"
Amal On
For X=1 To 320
   Bob 1,X,,
   Wait Vbl
Next X
```

Whenever a bob is moved, the area underneath is replaced in its original position, producing an identical effect to the equivalent SPRITE command.Unlike STOS on the ST, each object is allocated its own individual storage area.This reduces the amount of memory used by bobs, and improves the overall performance dramatically. Due to the Blitter, of course, there's no real comparison between STOS sprites and AMOS bobs.

Although the BOB command works fine for small numbers of bobs, there's an annoying flicker when you try to use more than three or four objects on the screen at once. This happens because the bobs are updated at the same time as the screen. You can therefore see the bobs while they are being drawn which results in an unpleasant shimmering effect.

One alternative for improving the quality of your animations is to just limit your bobs to the bottom quarter of the screen. Since bobs are redrawn extremely quickly, the updates can often be completed before the lower part of the screen has been displayed. This provides you with acceptably smooth movements while consuming very little memory, so it's a useful trick if you're running short of space. See **EXAMPLE 12.1**.

Obviously this cannot be seen as a serious solution to such a glaring problem. So before you throw away your copy of AMOS Basic in disgust, you'll be relieved to hear that there's a simple way of eliminating this flicker completely, even when you're using dozens of bobs anywhere on the screen:

DOUBLE BUFFER *(Create a double screen buffer)*

DOUBLE BUFFER

The DOUBLE BUFFER command creates a second invisible copy of the current screen. All graphics operations, including bob movements, are now performed directly in this *logical screen*, without disturbing your TV picture in the slightest. Once the image has been redrawn, the logical screen is displayed, and the original *physical* screen becomes the new logical screen. The entire process now cycles continuously, producing a rock solid display even when you are moving hundreds of bobs around on the screen at once.

The entire procedure is performed automatically by AMOS Basic, so after you've executed this instruction you can forget about it completely. Note that since the hardware sprites are always displayed using the current physical screen, this system will have absolutely no effect on any existing sprite animations.

Double buffering works equally well in all of the AMIGA's graphics modes. It can even be used in conjunction with dual playfields. But be warned: Double buffering doubles the amount of memory used by your screens. If you attempt to double buffer too many screens, you'll quickly run out of memory. See **EXAMPLE 12.2**.

In practice, double buffering is an incredibly useful technique, which can be readily exploited for most types of games. It has seen service in the vast majority of commercial games, including Starglider – that's why it's such an integral part of AMOS Basic. A detailed explanation of this process can be found in the SCREENS Chapter. Also see the SCREEN SWAP and AUTOBACK commands.

SET BOB *(Set drawing mode of bob)*

SET BOB n,back,planes,minterms

The SET BOB command changes the drawing mode used to display a blitter object on the screen. *n* is the number of the bob you wish to affect.

back chooses the way the background underneath your bob will be redrawn. There are three possibilities:

- A value of zero indicates that the area underneath your bob should be saved in memory. The old image data is automatically replaced when the bob is moved, resulting in a smooth movement effect.
- If the *back* parameter is positive then the original background will be discarded altogether, and the area behind the bob will be filled with colour *back-1*. This is ideal for moving bobs over a solid block of colour such as a clear blue sky, as it's much faster than the standard drawing system.
- Turn off the redrawing process completely by loading *back* with a negative value such as -1. You can now deactivate the automatic updating process using BOB UPDATE, and manually move your bobs with a call to BOB DRAW. This allows you to regenerate the screen background using your own customised drawing routines.

planes is a bit-map which tells AMOS which screen planes your bob will be drawn in. As you may know, the Amiga's screen is divided up into a number of separate bit-planes. Each plane sets a single bit in the final colour which is displayed on the screen.

The first plane is represented by bit one, the second by bit two and so on. Normally the bob is drawn in all the bit-planes in the current screen mode. This corresponds to a bit-pattern of %111111.

By changing some of these bits to zero, you can omit selected colours from your bobs when they are drawn. This can be used to generate a number of intriguing screen effects.

minterms selects the blitter mode used to draw your bobs on the screen. A full description of the available drawing modes can be found in the section on SCREEN COPY. *minterm* is usually set to one of two values:

%11100010	If the bob is used with a mask
%11001010	If NO MASK has been set

Feel free to experiment with the various combinations. There's no danger of crashing your Amiga if you make a mistake. Advanced Amiga users may find the following information useful.

Blitter source	Purpose
A	Blitter Mask
B	Blitter object
C	Destination screen

Note that you are recommended to use SET BOB **before** displaying your bobs on the screen. If you don't, the Amiga won't crash, and you won't get an error message, but your screen display may be corrupted.

NO MASK *(Remove blitter mask)*

NO MASK [n]

As a default, a blitter mask is automatically created for every bob you display on the screen. This mask is combined with the screen background to make colour zero transparent. It's also used by the various collision detection commands.

The NO MASK command removes this mask, and forces the entire image to be drawn on the screen. Any parts of the image in colour zero will now be displayed directly over the existing background.

n is the image number whose mask is to be removed. This mask should never be erased if the image is active on the screen, otherwise the associated bob will be corrupted. If you must remove the mask in this way, it's important to deactivate the relevant bobs with BOB OFF first. Here's an example:

```
Centre "Click mouse button to remove mask"
Double Buffer : Load "AMOS_Data:Sprites/Octopus.abk" : Get Sprite palette
Do
   Bob 1,X Screen(X Mouse),Y Screen(Y Mouse),1
   If Mouse Click Then Bob Off : No Mask 1
Loop
```

See MAKE MASK

AUTOBACK *(Set automatic screen copying mode)*

AUTOBACK n

When you are using a double buffered screen, it's essential to synchronize your drawing operations with the movements of your blitter objects. Remember that each double buffered screen consists of two separate displays. There's one screen for the current picture, and another for the image whilst it's being constructed. If the background underneath a bob changes while it's being redrawn, the contents of these screens will be different, and you'll get an intense and annoying flickering effect.

The unique AMOS AUTOBACK system provides you with a perfect solution to this problem. It allows you to generate your graphics in any one of three graphics modes, depending on the precise requirements of your program. Just for a change, we'll list these options in reverse order.

AUTOBACK 2 *(Automatic mode – default)*
In this mode, all drawing operations are automatically combined with the bob updates. So anything your draw on the screen will be displayed directly underneath your bobs, as if by magic. The principles behind this system can be demonstrated by the following code.

```
Rem Draw on first screen
Rem Remove Bobs
Bob Clear
Rem Perform Graphics operation
Plot 150,100 : Rem This can be anything you wish
Bob Draw : Rem Redraw Bobs
Screen Swap : Rem Next Screen
Wait Vbl
Rem Draw object on next screen
Bob Clear
Plot 150,100 : Rem Perform you operation a second time
Bob Draw
Screen Swap : Rem Get Back to first screen
Wait Vbl
```

As you can see, all screen updates are performed exactly twice. There's one operation for both the logical and the physical screen. See **EXAMPLE 12.3** for a demonstration.

One obvious side effect, is that your graphics now take twice as long to be drawn. Furthermore, the program will be halted by at least 2 vertical blanks, (or 2/50th of a second), every time you output something to the screen. This may cause annoying delays in the execution of critical activities such as collision detection.

AUTOBACK 1 *(Half-automatic mode)*
Performs each graphical operation in both the physical and logical screens. Absolutely no account is taken of your blitter objects, so you should only use this system for drawing outside the current playing area.

Unlike the standard mode, there's no need to halt your program until the next vertical blank. Mode 1 is therefore ideal for objects such as control panels or hi-score tables, which need to be updated continually during the game.

AUTOBACK 0 *(Manual mode)*
Stops the AUTOBACK system in it's tracks. All graphics are now output straight to the *logical* screen at the maximum possible speed. You should use this option if you need to repeatedly redraw large sections of your background screen during the course of a game. This will allow you to safely perform your collision detection routines at regular intervals, without destroying the overall quality of the animation effects. Here's a typical program loop for you to examine.

```
Bob Update Off : Rem Kill Automatic screen updates
Repeat
Screen Swap
```

```
Wait Vbl
Rem Erase your Bobs from the screen
Bob Clear
Rem
Rem Now redraw any of your graphics which have changed.
Rem Perform your game routines (Collision detection etc...)
Rem Update your Bob images
Rem
Bob Draw
Rem Swap Physical and logical screens
Until WIN : Rem Continue until the game has finished
```

Note that this procedure will **only** work if there's a smooth progression from screen to screen. It's entirely up to you to keep the contents of the physical and logical screens in step with each other. An example of this technique can be found in **EXAMPLE 12.4**

Supposing for instance, you wanted to display a bob over a series of random blocks. You might try to use a routine like :

```
                     Octopus.Abk
Load "AMOS_DATA:~~Sprites/Sprites.abk~~" : Flash Off : Get Sprite Palette
Double Buffer : Cls 0 : Autoback 0 : Update Off
Bob 1,160,100,1
Do
  Bob Clear
  X=Rnd(320)+1 : Y=Rnd(200)+1 : W=Rnd(80)+1 : H=Rnd(50)+1 : I=Rnd(15)
  Ink I : Bar X,Y TO X+W,Y+H
  Rem This would normally be a call to your collision detection routine
  Bob Draw
  Screen Swap : Wait Vbl
Loop
```

But since there's no relationship between the physical and logical screens, the display will now flick continuously from screen to screen. To overcome this problem, you'll need to mimic the original AUTOBACK system like so:

```
Load "AMOS_DATA:Sprites/Sprites.abk" : Flash Off : Get Sprite Palette
Double Buffer : Cls 0 : Autoback 0 : Update Off
Bob 1,160,100,1
Do
  Rem Update first screen
  Screen Swap : Wait Vbl
  Bob Clear
  X=Rnd(320)+1 : Y=Rnd(200)+1 : W=Rnd(80)+1 : H=Rnd(50)+1 : I=Rnd(15)
  Ink I : Bar X,Y To X+W,Y+H
  Bob Draw
  Rem Update second screen
  Screen swap : Wait Vbl
  Bob Clear
  Ink I : Bar X,Y To X+W,Y+H
```

```
   Bob Draw
Loop
```

The two screens are now updated with exactly the same information, and the display remains as steady as a rock, even though there's a great deal of activity going on in the background.

Autoback can be safely used at any point in your program. So it's perfectly possible to use separate drawing methods for the different parts of your screen. It's also totally compatible with all graphics operations including Blocks, Icons, and Windowing.

Bob control commands

BOB UPDATE *(Control bob movements)*

BOB UPDATE [ON/OFF]

Normally all bobs are updated once every 50th of a second using a built-in interrupt routine. Although this is convenient for most programs, there are some applications which require much finer control over the redrawing process.

BOB UPDATE OFF turns off the bob updates and deactivates all automatic screen switching operations if they've been selected. You may now redraw your bobs at the most appropriate point in your program using the BOB UPDATE command. This is ideal when you are animating a large number of objects as it enables you to move your bobs into position before drawing them on the screen. Inevitably this results in far smoother movements in your game.

One word of warning: The bob updates will only occur at the **next** vertical blank. Also note that BOB UPDATE will always redraw the bobs on the current logical screen, so if you forget to use the SCREEN SWAP command, nothing will apparently happen.

BOB CLEAR *(Remove all the bobs from the screen)*

BOB CLEAR

The BOB CLEAR instruction removes all active bobs from the screen, and redraws the background regions underneath. It's intended for use with BOB DRAW to provide an alternative to the standard BOB UPDATE command.

BOB DRAW *(Redraw bobs)*

BOB DRAW

Whenever the bobs are redrawn on the screen, the following steps are automatically performed:

1. All active bobs are removed from the LOGICAL screen and the background regions are replaced. This step is performed by BOB CLEAR.
2. A list is made of all bobs which have moved since the previous update.
3. The background regions under the new screen coordinates are saved in memory.

4. All active bobs are redrawn at their new positions on the logical screen.
5. If the DOUBLE BUFFER feature has been activated, the physical and logical screens are now swapped.

The BOB DRAW command performs steps 2 to 4 of this process directly. Supposing you wished to create a screen scrolling arcade game. In this situation, it would be absolutely vital for your scrolling operations to be perfectly synchronized with movement effects. If the aliens were to move while the scrolling was taking place, their background areas would be redrawn at the wrong place. This would totally corrupt your display, and would result in a hopeless jumble on the screen. Load **EXAMPLE 12.5** from the MANUAL folder for a demonstration of this process:

=X BOB *(Get X coordinate of bob)*

x1=X BOB(n)

Returns the current X coordinate of bob number *n*. This coordinate is measured relative to the current screen. See also Y SPRITE, X MOUSE and Y MOUSE .

=Y BOB *(Get Y coordinate of bob)*

y1=YBOB(n)

Y BOB complements the X BOB command by returning the Y coordinate of bob number *n*. The value will be returned using normal screen coordinates. See also X BOB, X SPRITE, Y SPRITE, X MOUSE and Y MOUSE.

=I BOB *(Return current image of a bob)*

Image=I Bob(n)

This function returns the current image number being used by bob n. A value of zero will be reported if the bob is not displayed.

LIMIT BOB *(Limit a bob to a rectangular region of the screen)*

LIMIT BOB [n,] x1,y1 TO x2,y2

This command restricts the visibility of your bobs to a rectangular screen area enclosed by the coordinates *x1,y1 to x2,y2* measured relative to the current screen. The x coordinates are rounded up to the nearest 16–pixel boundary. Note that the width of this region must always be greater than the width of your bobs, otherwise you'll get an *illegal function call* error.

If it's included, *n* specifies the number of a single bob which is to be affected by this instruction, otherwise **all** bobs will be restricted. You can restore the visibility limit to the entire screen by typing:

```
Limit Bob
```

GET BOB *(Load a section of the screen into the sprite bank)*

GET BOB [s,] i,x1,y1 To x2,y2

This instruction is identical to the GET SPRITE command. It grabs an image into the sprite bank from the current screen.

x1,y1 to *x2,y2* are the coordinates of the top and bottom corners of the rectangular area to be grabbed.

i specifies the image number which is to be loaded with this area. *s* selects an optional screen number from which the image is to be taken. See GET SPRITE for more details. For example:

```
Load Iff "AMOS_DATA:IFF/AMOSPIC.IFF"
Get Sprite 1,0,64 To 320,164
Clw
Bob 1,0,0,1
```

A larger example can be found in **EXAMPLE 12.6.**This loads a picture into memory and allows you to grab a bob into the sprite bank from any section of the screen. See HOT SPOT and MASK.

PUT BOB *(Fix a copy of a bob onto the screen)*

PUT BOB n

This is the exact opposite of the previous GET BOB command. The action of PUT BOB is to place a copy of bob number *n* at its present position on the screen. It works by preventing the background underneath the bob from being redrawn during the next vertical blank period. In order to synchronise the bob updates with the screen display, you should always follow this command with a WAIT VBL instruction.

Note that after this instruction has been performed, the original bob may be moved or animated with no ill effects.

PASTE BOB *(Draw an image from the sprite bank on the screen)*

PASTE BOB x,y,i

The PASTE BOB command draws a copy of image number *i* at **screen** coordinates *x,y*. Unlike PUT BOB this image is drawn on the screen immediately, and all the normal clipping rules are obeyed. See PASTE ICON.

BOB OFF *(Remove a bob from the display)*

BOB OFF [n]

Occasionally, you may wish to remove certain bobs from the screen altogether. The BOB OFF command erases bob number *n* from the screen and terminates any associated animations. If *n* is omitted, all bobs will be removed by this instruction. Try the following:

```
Load "AMOS_DATA:Sprites/Octopuss.abk"
Get Sprite Palette
Bob 2,110,110,2
Wait Key
Bob Off 2
Direct
```

13: Object control

In this section you will find out how the various objects generated using the sprite and bob commands can be controlled from within an AMOS Basic program. The topics under discussion include collision detection, using the mouse cursor and reading the joystick.

The mouse pointer

The mouse cursor provides the games programmer with a valuable alternative to the standard joystick. With the CHANGE MOUSE command you can replace the mouse with an image in the current sprite bank. There's also a group of instructions which allow you to determine both the position and status of this mouse at any time. These include the X MOUSE, Y MOUSE and MOUSE KEY instructions.

HIDE *(Remove mouse pointer from the screen)*

HIDE [ON]

This command removes the mouse pointer from the screen completely. A count of the number of occasions you have called this function is kept internally by the system. This needs to be matched by an equal number of SHOW instructions before the pointer will be returned on the screen.

There's also another version of this instruction which can be accessed with HIDE ON. This ignores the count and **always** hides the mouse, no matter how many times you've called the SHOW command.

Note that HIDE only makes the mouse pointer invisible. It has no effect on any other AMOS commands, so you can still use the X MOUSE and Y MOUSE functions to read the coordinates of the mouse as normal.

SHOW *(Activate the mouse pointer)*

SHOW [ON]

This returns the mouse pointer to the screen after a HIDE instruction. As a default, a count is kept of the number of previous HIDE commands. If the number of SHOWs is less than the number of HIDEs the pointer will stay invisible. To ignore this count and activate the mouse immediately, use the SHOW ON command instead.

CHANGE MOUSE *(Change the shape of the mouse pointer)*

CHANGE MOUSE m

This allows you to change the shape of the mouse at any time. Three mouse patterns are provided as standard. These can be assigned using the numbers 1 - 3 as follows:

M	Shape
1	Arrow pointer (Default)
2	Crosshair
3	Clock

If you specify a value of *m* greater than 3, this is assumed to refer to an image stored in the sprite bank. The number of this image is determined using the expression *I=m-3.* So image number 1 would be installed by a value of 4, and image 2 would be loaded by a 5.

In order to use this option, your sprite image must be exactly 16 pixels wide and have no more than four colours. However there's no such limit to the height of your image. See also HIDE, SHOW, X MOUSE, Y MOUSE, MOUSE KEY, LIMIT MOUSE.

=MOUSE KEY *(Read status of mouse buttons)*

k=MOUSE KEY

The MOUSE KEY function enables you to quickly check whether one or more of the mouse keys have been pressed. It returns a bit-pattern which holds the current status of the mouse buttons.

This bit-pattern has the following format:

Bit 0	Set to 1 if the LEFT button pressed, otherwise zero
Bit 1	Status of RIGHT button using the same format
Bit 2	Status of THIRD button if available

Example:

```
Curs Off
Do
  Locate 0,0
  M=Mouse Key: Print “Bit Pattern “;Bin$(M,8);” Number”,M
Loop
```

=MOUSE CLICK *(Check for a mouse click)*

c=MOUSE CLICK

The MOUSE CLICK function checks whether the user has “clicked” on a mouse button. It returns a bit-pattern in the format:

Bit 0	One shot test for LEFT button
Bit 1	One shot test for RIGHT Button
Bit 2	One shot test for THIRD Button if available

One shot tests are only set to 1 when the mouse key has just been pressed. These bits are automatically reset to zero after they've been tested once. So they will only check for a single key press at a time. Here's an example:

```
Curs Off
Do
    M=Mouse Key
    If M<>0 Then Print "Bit Pattern ";Bin$(M,8);" Number",M
Loop
```

=X MOUSE= *(Get /set the X coordinate of the mouse pointer)*

x1=X MOUSE

X MOUSE returns the current X coordinate of the mouse pointer in *hardware* notation.

You can also use this function to **move** the mouse on to a specific screen position. This can be achieved by assigning X MOUSE with a value, just like a Basic variable, for example:

```
X Mouse=150 : Rem Moves mouse to hardware coordinate 150, Y coord is not affected
```

=Y MOUSE= *(Get/set the Y coordinate of the mouse pointer)*

y1=Y MOUSE

This function returns the Y coordinate of the mouse pointer. As with X MOUSE the result uses *hardware* coordinates rather than the more normal screen coordinates. Y MOUSE can also be used to reposition the mouse pointer on the screen. Simply load the new coordinate into Y MOUSE like so:

```
Y Mouse=100
```

For an example of the X MOUSE and Y MOUSE functions load **EXAMPLE 13.1** from the MANUAL folder.

LIMIT MOUSE *(Limit mouse to a section of the screen)*

LIMIT MOUSE x1,y1 TO x2,y2

Restricts mouse movements to the rectangular area defined by the hardware coordinates *(x1,y1)* and *(x2,y2)*. *x1,y1* denotes the top left hand corner of this box and *x2,y2* the point diagonally opposite. Note that unlike LIMIT BOB, the mouse is completely trapped inside this zone and cannot be moved beyond it. Load up **EXAMPLE 13.2** from the manual folder for a demonstration of this command. Simply use the instruction with no parameters to restore the mouse to the full screen area.

```
Limit Mouse
```

Reading the joystick

AMOS Basic includes six functions which allow you to immediately check the movements of a joystick inserted in either of the available sockets.

=JOY *(Read joystick)*

d=JOY(j)

This function returns a binary number which represents the current status of a joystick in port number *j*. Normally your joystick will be placed in the left socket (number 1). However you can remove the mouse from the right-hand socket and replace it with a joystick. This can be accessed using port number zero.

The state of the joystick can be read by inspecting the pattern of binary bits in the result. Each bit indicates whether a specific action has been performed by the user. If a bit is set to one then the test has proved positive and the joystick has been moved in the appropriate direction.

Here's a list of the various bits and their meanings:

Bit number	**Significance**
0	Joystick moved up
1	Joystick moved down
2	Joystick moved left
3	Joystick moved right
4	Fire button pressed

Don't worry if you are not familiar with this binary notation. You can also access each of the directions individually with the functions JLEFT, JRIGHT, JUP, JDOWN and FIRE. See **EXAMPLE 13.3** for a demonstration of this command.

=JLEFT *(Test joystick movement left)*

x=JLEFT(j)

JLEFT returns a value of -1 (True) if the joystick in port *j* has been pulled to the left, otherwise 0 (False) is returned. The joystick sockets are numbered from right to left, starting at zero. So the default joystick socket is accessed using port number 1. For example:

```
Do
 If Jleft(1) Then Print "LEFT"
Loop
```

=JRIGHT *(Test joystick movement right)*

x=JRIGHT(j)

JRIGHT tests joystick number *j* and returns -1(True) if has been moved right, otherwise 0 (False). See JLEFT, JUP, JDOWN.

=JUP *(Test joystick movement up)*

x=JUP(j)

JUP returns -1 if joystick *j* has been pushed up, otherwise 0. See JRIGHT, JLEFT, JDOWN.

=JDOWN *(Test joystick movement down)*

x=JDOWN(j)

The JDOWN function returns the value -1 if a joystick has been pulled down, otherwise it returns 0. See JRIGHT, JLEFT, JUP.

=FIRE *(Test fire button state)*

x=FIRE(j)

This function only returns a value of -1 if the fire button on joystick number *j* has been pressed. See JUP, JDOWN, JLEFT, JRIGHT, JOY .

Detecting collisions

If you're writing an arcade game it's vital to be able to accurately check for collisions between the various objects on the screen. AMOS Basic includes five powerful functions which allow you to perform these tests quickly and easily.

Detecting collisions with a sprite

SPRITE COL *(Detect collisions between two hardware sprites)*

c=SPRITE COL(n [,s TO e])

This provides you with a simple way of testing to see whether two or more sprites have collided on the screen. The number *n* refers to an active hardware sprite which is to be checked for a collision. If a collision has occurred a value of -1 (true) will be returned, otherwise the result will be set to 0 (false).

The standard form of this function checks for all collisions. But you can also test a whole group of sprites using an extended version of the command:

c=SPRITE COL n,s TO e

The above instruction checks for collisions between sprite *n* and sprites *s* to *e* (inclusive). Once you've detected a collision, you can then get the individual sprite numbers which have collided using the COL function.

Note that in order to use this function, you'll need to create a *sprite mask* with the MASK command first, otherwise your collisions will not be detected. A detailed example of this command can be found in **EXAMPLE 13.4.**

Detecting collisions with a bob

BOB COL *(Detect collisions between two Blitter objects)*

c=BOB(n [,s TO e])

The BOB COL function checks bob number *n* for a collision with another bob. If a collision has been detected, the value returned in *c* will be set to -1 (true), otherwise it will be loaded with 0(false).

Normally the command will check for all collisions, but you can specify a collection of bobs to be tested using the optional range parameters *s* to *e*. The status of these bobs can be individually examined with the COL command. See **EXAMPLE 13.5**.

Collisions between bobs and sprites

In AMOS Basic you're not just limited to detecting collisions between the same types of objects. It's also possible to check for any combination of sprites and bobs on the screen.

SPRITEBOB COL *(Test for a collision between sprites and bobs)*

c=SPRITEBOB COL(n [,s TO e])

This function checks for a collision between **sprite** *n* and one or more **bobs**. The value of *c* will be either -1 if a collision has been discovered, or 0 if there have been no collisions. The starting and ending points specify that collisions will only be detected between the bobs *s* to *e*. If they are not included then all active bobs will be tested by this function. See the COL command for more details.

Warning! Collision detection between a sprite and a bob is only possible on a low resolution screen. In Hires mode, the pixel sizes used for bobs and sprites are totally different, and the results from this function will be unreliable.

BOBSPRITE COL *(Test for collision between bobs and sprites)*

c=BOBSPRITE COL(n [,s TO e])

The BOB SPRITE COL function checks for collisions between a single bob and several sprites. If the test is successful a value of -1 will be returned, otherwise it will return a 0. The optional range parameters list a group of sprites to be tested from *s* to *e*. If it is omitted then all the currently active sprites will be checked by this function.

Note that BOB SPRITE COL **only** works with low resolution screens. So don't attempt to use it in conjunction with Hires or you'll get an unpredictable result. A full demonstration of this command can be found in **EXAMPLE 13.6** in the MANUAL folder.

=COL *(Test the status of a sprite or bob after a collision detection instruction)*

c=COL(n)

The COL array holds the status of all the objects which have been previously tested by the collision detection functions.

Each object you have checked is associated with one element in this array. This element will be loaded with -1 if a collision has been detected with object number *n*, or 0 if it has not. The numbering system is simple: The first element in the array holds the status of object number 1, the second represents object number 2, and so on. See **EXAMPLE 13.7**.

If you are using the SPRITE COL or BOB SPRITE COL instructions then the objects will be hardware sprites, otherwise they will be Blitter objects. In order to avoid confusion, it's sensible to call this instruction immediately after the relevant detection command.

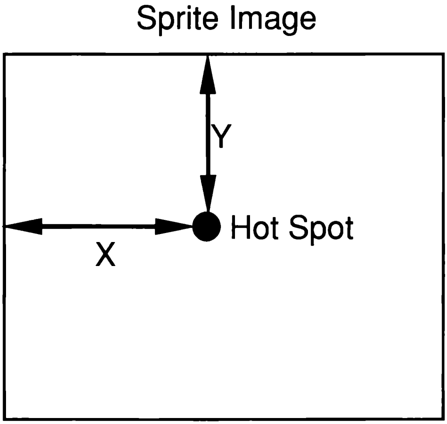
HOT SPOT *(Set the hot spot for an image in the sprite bank)*

HOT SPOT image,x,y
HOT SPOT image,p

This command sets the hot spot of an image stored in the current sprite bank. The hot spot of the object is used as a reference point for all coordinate calculations. There are two versions of this instruction.

HOT SPOT image,x,y

x and y coordinates measured from the top left hand corner of the image. These coordinates will be added to the sprite or bob coordinate to position an object precisely on the screen.



Note that it's perfectly legal for the hot spot to lie outside the actual image.

HOT SPOT image,p

This is a short form of the instruction which moves the hot spot to one of nine predefined positions. The positions are shown in the diagram below where the centre point of the image is represent by a value of $11.

$00	$10	$20
$01	$11	$21
$02	$12	$22

See **EXAMPLE 13.8**.

MAKE MASK *(Make a mask around an image for collision detection)*

MAKE MASK [n]

MAKE MASK defines a mask around image number *n* in the sprite bank. This is used by all the AMOS Basic collision detection commands. You should therefore create a mask for every object you wish to check. If you omit the image number *n*, then a mask will be generated for each image in the sprite bank. This may take a little time.

It's important to note that masks are generated automatically when a bob is first drawn on the screen. This might cause a significant delay in the running of your program, so it's worthwhile placing an explicit call to MAKE MASK during your initialisation procedure.

Collisions with rectangular blocks

AMOS Basic includes a number of functions which allow you to quickly check whether a sprite or bob has entered a rectangular region of the screen.

These *screen zones* are especially useful for collision detection in rebound games such as Arkanoid as each block can be assigned its own individual screen zone. You can also use zones to construct the buttons and switches needed for control panels and dialogue boxes.

RESERVE ZONE *(Reserve space for a detection zone)*

RESERVE ZONE [n]

RESERVE ZONE allocates enough memory for exactly *n* detection zones. This command should always be used before defining a zone with SET ZONE.

The only limit to the number of zones is the amount of available memory, so it's perfectly feasible to define hundreds or even thousands of zones in one of your programs.

To erase the current zone definitions and restore the memory back to the main program, simply type RESERVE ZONE with no parameters.

SET ZONE *(Set a zone for testing)*

SET ZONE z,x1,y1 TO x2,y2

Defines a rectangular zone which can be subsequently tested using the various ZONE commands. *z* specifies the number of the zone to be created and *x1,y1* and *x2,y2* input the coordinates of the top left and bottom right hand corners of the rectangle.

Before using this instruction you'll need to reserve some space for your zones with RESERVE ZONE.

See ZONE, RESET ZONE , RESERVE ZONE, ZONE$.

=ZONE *(Return the zone under the requested screen coordinates)*

t=ZONE([s],x,y)

ZONE returns the number of the screen zone at the graphic coordinates *x,y.* Normally the coordinates are relative to the current screen – you can also include an optional screen number *s* in this function.

After ZONE has been called, *t* will hold either the number of the zone at the specified coordinates or a value of 0 (false).

Note that ZONE only returns the first zone at these coordinates – it won't detect any other zones which lie inside this region.

In order to demonstrate this command we've included two examples programs in the MANUAL folder. These can be found in the files **EXAMPLE 13.9** and **EXAMPLE 13.10**. Feel free to modify them for use in your own games.

It's possible to use this function in conjunction with the X BOB and Y BOB functions to detect whether a bob has entered a specific screen zone. This can be accomplished using the following code:

```
X=Zone(X Bob(n),Y Bob(n))
```

See HZONE, SET ZONE, RESET ZONE, X BOB, Y BOB.

=HZONE *(Return the zone under the requested hardware coordinates)*

t=HZONE([s],x,y)

HZONE is almost identical to ZONE except that the screen position is now measured in *hardware* coordinates. You can therefore use this function to detect when a hardware sprite enters one of your screen zones. For example:

```
X=Hzone(X Sprite(n),Y Sprite(n))
```

A demonstration of this command can be found in **EXAMPLE 13.11.** See ZONE,MOUSE ZONE,SET ZONE,RESERVE ZONE,ZONE$.

=MOUSE ZONE *(Check whether the mouse pointer has entered a zone)*

x=MOUSE ZONE

The MOUSE ZONE function returns the number of the screen zone currently occupied by the mouse pointer. It's equivalent to the line:

```
X=Hzone(X mouse,Y mouse)
```

See ZONE, HZONE, SET ZONE, ZONE$.

RESET ZONE *(Erase a zone)*

RESET ZONE [z]

This command permanently deactivates any of the zones created by SET ZONE. If the optional zone number *z* is included then only this zone will be reset, otherwise all the zones will be affected. Note that RESET ZONE only erases the zone definitions, it does not return the memory allocated by RESERVE ZONE.

Bob priority

PRIORITY ON/OFF *(Change between priority modes)*

PRIORITY ON/OFF

Each bob is assigned a *priority* value ranging from 0-63. AMOS Basic uses this number to decide which order the objects should be displayed on the screen. As a rule, any bob with the highest priority will always be displayed in front of any objects with a lower priority. The priority value is taken directly from the number of a Bob.

You should remember this fact when assigning numbers to your bobs. The choice of number can have wide ranging effects on the appearance of your objects on the screen.

In addition to the standard system, it's also possible to arrange the bobs according to their position on the screen. PRIORITY ON assigns the greatest priority values to the bobs with the highest Y coordinates. This allows you to create a useful illusion of perspective in your games. Look at the example below.

```
Load “AMOS_DATA/Sprites/Monkey_right.abk” : Cls : Flash Off : Get Sprite Palette
Double Buffer
Priority Off : Rem Set Normal Mode
Bob 1,160,100,2 : Bob 2,0,74,2 : Bob 3,320,128,2
Channel 2 To Bob 2 : Channel 3 To Bob 3
Amal 2,” Loop:  M  320,0,320 ;  M  -320,0,320 ; Jump Loop”
Amal 3,” Loop:  M  -320,0,320 ;  M  320,0,320 ; Jump Loop”
Amal On
Wait Key
Priority On : Rem Set Y Mode
Wait Key
```

Normally, both moving bobs pass below the object in the centre. When you change the priority system with a call to PRIORITY ON, the bobs are now ranked in order of their increasing Y coordinates. So bob three moves above bob one while at the same time, bob two passes smoothly behind it.

HINT: It's usually best to position the Hot Spot of the sprite at its base. This is because the Y coordinates used by this command relate to the position of the Hot Spot on the screen. Also notice that the PRIORITY OFF instruction can be utilised to reset the priority back to normal.

Miscellaneous commands

UPDATE *(Change automatic sprite/bob updates)*

UPDATE [ON/OFF]

Normally any objects you draw on the screen will be automatically redisplayed whenever they are animated or moved. This feature can be temporarily halted using the UPDATE OFF command. When the updates are not active the SPRITE, BOB and AMAL commands apparently have no effect. Actually, all your animations are working correctly – it's just that the results are not being displayed on the screen. You can force this redrawing operation at any time using the UPDATE command. Here are the three different forms of the UPDATE instruction:

```
UPDATE OFF
```

Turns off the automatic updating of both the sprites and bobs. Any movements or animations will appear to be suspended.

```
UPDATE
```

Redraws any sprites which have changed their original positions.

```
UPDATE ON
```

Returns the sprite updating to normal. See **EXAMPLE 13.12**. See UPDATE EVERY, SYNCHRO, SPRITE UPDATE, BOB UPDATE.

14: AMAL

If you wish to generate the smooth movement required in an arcade game, it's necessary to move each object on the screen dozens of times a second. This is a real struggle even in machine code and it's way beyond the abilities of the fastest version of Basic.

AMOS sidesteps this problem by incorporating a powerful animation language which is executed independently of your Basic programs. This is capable of generating high speed animation effects which would be impossible in standard Basic.

The **AM**OS **A**nimation **L**anguage (AMAL) is unique to AMOS Basic. It can be used to animate anything from a sprite to an entire screen at incredible speed. Up to 16 AMAL programs can be executed simultaneously using interrupts.

Each program controls the movements of a single object on the screen. Objects may be moved in complex predefined attack patterns, created from a separate editor accessory. You can also control your objects directly from the mouse or joystick if required.

The sheer versatility of the AMAL system has to be seen to be believed. Load up **1** from the MANUAL folder for a complete demonstration.

AMAL principles

AMAL is effectively just a simple version of Basic which has been carefully optimised for the maximum possible speed. As with Basic, there are instructions for program control (Jump), making decisions (If) and repeating sections of code in loops (For...Next). The real punch comes when an AMAL program is run. Not only are the commands lightning fast but all AMAL programs are **compiled** before run-time.

AMAL commands are entered using short keywords consisting of one or more capital letters. Anything in lowercase is ignored completely. This allows you to pad out your AMAL instructions into something more readable. So the M command might be entered as Move or the L instruction as Let.

AMAL instructions can be separated by practically any unused characters including spaces. You can't however, use the colon ":" for this purpose, as it's needed to define a label. We advise you use a semi-colon ";" to separate commands to avoid possible AMAL headaches.

There are two ways of creating your AMAL programs. The first is to produce your animation sequences with the AMAL accessory program and save them into a memory bank or you can define your animations inside AMOS Basic using the AMAL command. The general format of this instruction is:

```
AMAL n,a$
```

n is the identification number of your new AMAL program. As a default all programs are assigned to the relevant hardware sprite. So the first AMAL program controls sprite number one, the second sprite number two, and so on. You can change this selection at any time using a separate CHANNEL command. *a$* is a string containing a list of AMAL instructions to be performed in your program. Here's a simple example:

```
Load "AMOS_DATA:Sprites/Monkey_right.abk" : Rem Load some example sprites
Get Sprite Palette
Sprite 8,130,50,1 : Rem Place a sprite on the screen
```

```
Amal 8,"S: M 300,200,100 ; M -300,-200,100 J S" : Rem Define a small AMAL program
Amal On 8 : Rem Activate AMAL program number eight
Direct
```

The program returns you straight back to direct mode with the DIRECT command. Try typing a few Basic commands at this point. You can see the movement pattern continues regardless, without interfering with the rest of the AMOS system. Also note we have used sprite 8 to force the use of a computed sprite. All computed sprites from 8 to 15 are automatically assigned to the equivalent channel number by the AMAL system. So there's no need for any special initialisation procedures. Unless you wish to restrict the amount of hardware sprites it's safest to stick to just computed sprites in your programs. Notice how we've activated the AMAL program using the AMAL ON command. This has the format:

AMAL ON [prog]

prog is the number of a single AMAL program you wish to start. If it's omitted then all your AMAL programs will be executed at once.

AMAL tutorial

We'll now provide you with a guided tour of the AMAL system. This will allow you to slowly familiarise yourself with the mechanics of AMAL programs, without having to worry about too many technical details.

For the time being we'll be concentrating on sprite movements, but the same principles can also be applied to bob or screen animations.

Start off by loading some example sprites into memory. These can be found in the **Sprites** folder on the AMOS data disc. To get a directory of Sprite files type the following from the direct window:

```
Dir "AMOS_DATA:"
```

To load a sprite file, type a line like:

```
Load "AMOS_DATA:Sprites/octopus.abk"
```

Moving an object

As you would expect from a dedicated animation language, AMAL allows you to move your objects in a variety of different ways. The simplest of these involves the use of the Move command.

Move *(Move an object)*

M w,h,n

The M command moves an object *w* units to the right and *h* units down in exactly *n* movement steps. If the coordinates of your object were (X,Y), then the object would progressively move to X+W,Y+H.

Supposing you have a sprite at coordinates 100,100. The instruction M 100,100,100 would move it to 200,200. The speed of this motion depends on the number of movement steps. If *n* is large, then each individual sprite movement will be small and the sprite will move very slowly. Conversely, a small value for *n* results in large movement steps which jerk the sprite across the screen at high speed. Here are some examples of the move command:

```
Rem This moves an octopus sprite down the screen using AMAL
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
sprite 8,300,0,1
Amal 8,"M 0,250,50" : Amal On 8 : Wait Key

Rem This version moves an octopus sprite across the screen
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Sprite 9,150,150,1
Amal 9,"M 300,0,50" : Amal On 9 : Wait Key

Rem Moves octopus down and across the screen
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Sprite 10,150,150,1
Amal 10,"M 300,-100,50" : Amal On 10 : Wait Key

Rem Demonstrates multiple Move commands
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
M$="Move 300,0,50 ; Move -300,0,50"
Sprite 11,150,150,1
Amal 11,A$ : Amal On 11 : Wait Key
```

Notice how we've expanded M to Move in the above program. Since the letters "ove" are in lower case, they will be ignored by the AMAL system.

At first glance, Move is a powerful but unexciting little instruction. It's ideal for moving objects such as missiles, but otherwise it's pretty uninspiring.

Actually nothing could be further from the truth. That's because the parameters in the Move instruction are not limited to simple numbers. You can also use complex arithmetical expressions incorporating one of a variety of useful AMAL functions. Example:

```
Load "AMOS_DATA:Sprites\octopus.abk" : Get Sprite Palette : Sprite 12,150,150,1
Amal 12,"Move XM-X,YM-Y,32"
Amal On 12 : Wait Key
```

This smoothly moves computed sprite 12 to the current mouse position. X and Y hold the coordinates of your sprite, and XM and YM are functions returning the current coordinates of the mouse.

It's possible to exploit this effect in games like Pac-Man to make your objects *chase* the player's character. A demonstration of this procedure can be found in **2**.

The Move command can also be used to animate a whole screen. Here's a simple example:

```
Load Iff "AMOS_DATA:IFF/Frog_Screen.IFF",1
```

```
Channel 1 To Screen Display 1 : Rem Assigns AMAL program 1 to screen 1
Amal 1,”Move 0,-200,50 ; Move 0,200,50"
Amal On 1 : Direct
```

CHANNEL assigns an AMOS program to a particular object. We'll be discussing this command in detail slightly later, but the basic format is:

CHANNEL p To object n

p is the number of your AMAL program. Allowable values range from 0 to 63, although only the first 16 of these programs can be performed using interrupts.

object specifies the type of object you wish to control with your AMAL program. This is indicated using one of the following statements:

Sprite	(Values greater than seven refer to computed sprites)
Bob	(Blitter object)
Screen Display	(Used to move the screen display)
Screen Offset	(Hardware scrolling)
Screen Size	(Changes the screen size using interrupts)
Rainbow	(Animates a rainbow effect)

n is the number of the object to be animated. This object needs to be subsequently defined using the SPRITE, BOB or SCREEN OPEN instructions. Examples:

```
Channel 2 To Bob 1 : Rem Animate Bob 1 using AMAL program number 2
Channel 3 To Sprite 8 : Rem Assign channel three to a computed sprite
Channel 4 To Screen Display 0 : Rem Move default screen via AMAL
Channel 5 To Screen Offset 0 : Rem Change the screen offset within AMAL
```

Animation

Anim *(Animate an object)*

A n,(image,delay)(image,delay)...

The Anim instruction cycles an object through a sequence of images, producing a smooth animation effect. *n* is the number of times the animation cycle is to be repeated. A value of zero for this parameter will perform the animation continuously.

image specifies the number of an image to be used for each frame of your animation. *delay* determines the length of time this image is to be displayed on the screen, measured in units of a 50th of a second. Examples:

```
Load “AMOS_DATA:Sprites/octopus.abk” : Get Sprite Palette
Sprite 8,260,100,1
Amal 8, “A 0,(1,2)(2,2)(3,2)(4,2)”
Amal On 8 : Direct

Load “AMOS_DATA:Sprites/Monkey_right.abk” : Get Sprite Palette
```

```
Sprite 9,150,50,11
M$="Anim 12, (1,4)(2,4)(3,4)(4,4)(5,4)(6,4) ; "
M$=M$+"Move 300,150,150 ; Move -300,-150,75"
Amal 9,M$
Amal On 9
Direct
```

The second example combines a sprite movement with an animation. Notice how we've separated the commands with a semi-colon ";". This ensures that the two operations are totally independent of each other. Once the animation sequence has been defined, AMAL will immediately jump to the next instruction, and the animation will begin.

It's important to realize that Anim only works in conjunction with sprites and bobs. So it's not possible to animate entire screens with this command.

Simple Loops

Jump *(Redirects an AMAL program)*

J label

Jump provides a simple way of moving from one part of an AMAL program to another. *label* is the target of your jump, and must have been defined elsewhere in your current program.

All AMAL labels are defined using a single uppercase letter followed by a colon. Like instructions, you can pad them out with lower case letters to improve readability. Here are some examples:

```
S:
Swoop:
Label:
```

Remember that each label is defined using just a **single** letter. So S and Swoop actually refer to the **same** label! If you attempt to define two labels starting with an identical letter, you'll be presented with a *label already defined in animation string* error.

Each AMAL program can have its own unique set of labels. Its perfectly acceptable to use the identical labels in several different programs. Example:

```
Load "AMOS_DATA:Sprites/octopus.abk"
Get Sprite Palette
Rem Set up seven computed sprites down the screen
For S=8 To 20 Step 2
   Sprite S,200,(S-7)*13+40,1
Next S
Rem Create seven AMAL programs
For S=1 To 7
   Channel S To Sprite 6+(S*2)
   M$="Anim 0,(1,2)(2,2)(3,2)(4,2) ; Label: Move "+Str$(S*2)+",0,7 ; "
   M$=M$+"Move -"+Str$((6-2)*2)+",0,7 ; Jump Label"
   Amal S,M$
Next S
Rem Okay, now animate it all!
Amal On : Direct
```

This next example repeatedly moves a sprite to the current mouse position:

```
Load "AMOS_DATA:Sprite/octopus.abk"
Get Sprite Palette
Sprite 8,150,150,1
Amal 8,"R: Move XM-X,YM-Y,8 ; Pause; Jump R"
Amal On 8
```

Since AMAL commands are performed using interrupts, infinite loops could be disastrous. So a special counter is automatically kept of the number of jumps in your program. When the counter exceeds ten, any further jumps will be totally ignored by the AMAL system.

Note: if you rely on this system, and allow your programs to loop continually, you'll waste a great deal of the Amiga's computer power. In practice, it's much more efficient to limit yourself to just a single jump per VBL. This can be achieved by adding a simple PAUSE command before each Jump in your program. See PAUSE for more details.

Variables and expressions

Let *(Assigns a value to a register)*

L register=expression

The L instruction assigns a value to an AMAL register. The action is very similar to normal Basic, except that all expressions are evaluated strictly from left to right.

Registers are integer variables used to hold the intermediate values in your AMAL programs. Allowable numbers range between -32768 to +32767. There are three basic types of register:

Internal registers

Every AMAL program has its own set of 10 *internal* registers. The names of these registers start with the letter R, followed by one of the digits from 0 to 9 (R0-R9).

Internal registers are like the local variables defined inside an AMOS Basic procedure.

External Registers

External registers are rather different because they retain their values between separate AMAL programs. This allows you to use these registers to pass information between several AMAL routines. AMAL provides you with up to 26 external registers, with names ranging from RA to RZ.

The contents of any internal or external register can be accessed directly from your Basic program using the AMREG function (explained later).

Special Registers

Special registers are a set of three values which determine the status of your object.

X, Y contain the coordinates of your object. By changing these registers you can move your object around on the screen. Example:

```
Load “AMOS_DATA:Sprites/Frog_Sprites.abk” : Channel 1 To Bob 1
Flash Off : Get Sprite palette : Bob 1,0,0,1
Amal 1,”Loop: Let X=X+1 ; Let Y=Y+1; Pause; Jump Loop”
Amal On 1 : Direct
```

A stores the number of the image which is displayed by a sprite or bob. You can alter this value to generate your own animation sequences like so:

```
Load “AMOS_DATA:Sprites/Frog_Sprites.abk” : Get Sprite Palette : Flash Off
Channel 2 to Bob 1 : Bob 1,300,100,1
M$=”Loop: Let A=A+1 ; “
M$=M$+”For R0=1 To 5 ; Next R0 ; Jump Loop”
Amal 2,M$
Amal On 2 : Direct
```

The *For To Next* loop will be explained in more detail below. It is used here to slow down each change to Bob 1's image. When the *Next* of the loop is executed, AMAL won't continue until a vertical blank has occurred. Also note the use of “;” to separate the AMAL instructions – although a space will serve just as well.

Operators

AMAL expressions can include all the normal arithmetic operations, except MOD. You can also use the following logical operations in your calculations:

&	logical AND
|	logical OR

Note that it's not possible to change the order of evaluation using brackets “()” as this would slow down your calculations considerably and thus reduce the allowable time in the interrupt. Now for some more examples for you to type in:

```
Load “AMOS_DATA:Sprites/octopus.abk” : Hide
Get Sprite Palette
Sprite 8,X MOUSE,Y MOUSE,1
Amal 8,”Loop: Let X=XM ; Let Y=YM ; Pause ; Jump Loop”
Amal On 8
```

```
Load “AMOS_DATA:Sprites/octopus.abk” : Hide
Get Sprite Palette
Sprite 8,X MOUSE,Y MOUSE,1
Amal 8,”Anim 0,(1,4)(2,4)(3,4)(4,4) ; Loop: Let X=XM ; Let Y=YM ; Pause ; Jump Loop”
Amal On
```

The above examples effectively mimic the CHANGE MOUSE command. However this system is much more powerful as you can easily move bobs, computed sprites, or even screens using exactly the same technique.

Making Decisions

If *(Branch within an AMAL string)*

If test Jump L

This instruction allows you to perform simple tests in your AMAL programs. If the expression *test* is -1 (true) the program will jump to label *L,* otherwise AMAL will immediately progress to the next instruction. Note that unlike it's basic equivalent, you're limited to a single jump operation after the test.

It's common practice to pad out this instruction with lowercase commands like “then” or “else”. This makes the action of the command rather more obvious. Here's an example:

```
If X>100 then Jump Label else Let X=X+1
```

test can be any logical expression you like, and may include:

<>	Not equals
<	Less than
>	Greater than
=	Equals

Example:

```
Load “AMOS_DATA:Sprites/octopus.abk”
Get Sprite Palette
Sprite 8,130,50,1
C$=”Main:  If XM>100 Jump Test: “
C$=C$+” Let X=XM”
C$=C$+” Test: If YM>100 Jump Main “
C$=C$+” Let Y=YM Jump Main”
Amal 8,C$ : Amal On : Direct
```

A larger example can be found in **4** which allows you to control the position of a sprite using the joystick. This is actually quite crude and could be speeded up dramatically with the help of the AUTOTEST command. See AUTOTEST.

Warning! Don't try to combine several tests into a single AMAL expression using “&” or “|”. Since expressions are evaluated from left to right, this will generate an error. Take the expression: X>100|Y>100. This is intended to check whether X>100 OR Y>100. In practice, the expression will be evaluated in the following order:

X>100	May be TRUE or FALSE
|Y	OR result with Y

>100	Check if (Y>100|Y)>100)

The result from the above expression will obviously bear no relation to the expected value. Technically-minded users can avoid this problem by using boolean algebra. First assign each test to an single AMAL register like so:

```
Let R0=X>100; Let R1=Y>100
```

Now combine these tests into a single expression using "|" and "&" and use it directly in your If statement.

```
If R0 | R1 then Jump L ...
```

This may look a little crazy, but it works beautifully in practice.

For To Next *(Loop within AMAL)*

```
For reg=start To end
:   :
Next reg
```

This implements a standard FOR...NEXT loop which is almost identical to its Basic equivalent. These loops can be exploited in your programs to move objects in complex visual patterns. *reg* may be any normal AMAL register (R0-R9 or RA-RZ). However you can't use special registers for this purpose.

As with Basic, the register after the Next must match with the counter you specified in the For, otherwise you'll get an AMAL syntax error. Also note that the step size is always set to one. Additionally, it's possible to "nest" any number of loops inside each other.

Note that each animation channel will only perform a single loop per VBL. This synchronizes the effects of your loops with the screen display, and avoids the need to add an explicit Pause command before each Next.

Generating an attack wave for a game

These loops can be used to create some quite complex movement patterns. The easiest type of motion is in a straight line. This can be generated using a single For...Next loop like so:

```
Load “AMOS_DATA:Sprites/octopus.abk” : Get Sprite Palette
Sprite 8,130,60,1
C$="For R0=1 To 320 ; Let X=X+1 ; Next R0" : Rem Move Sprite from left to right
Amal 8,C$ : Amal On 8 : Direct
```

You can now expand this program to sweep the object back and forth across the screen.

```
Load “AMOS_DATA:Sprites/octopus.abk” : Get Sprite Palette
Sprite 8,130,60,1
C$="Loop: For R0=1 to 320 ; Let X=X+1 ; Next R0 ; “ : Rem Move sprite forward
```

```
C$=C$+"For R0=1 To 320 ; Let X=X-1 ; Next R0 ; Jump Loop" : Rem Move Sprite back
Amal 8,C$ : Amal On 8 : Direct
```

The first loop moves the object from left to right, and the second from right to left.

So far the pattern has been restricted to just horizontal movements. In order to create a realistic attack wave, it's necessary to incorporate a vertical component to this motion as well. This can be achieved by enclosing your program with yet another loop.

```
Load "AMOS_DATA:Sprites/octopus.abk"
Get Sprite Palette
Sprite 8,130,60,1
C$="For R1=0 To 10 ;"
C$=C$+"For R0=1 To 320 ; Let X=X+1 ; Next R0 ;" : Rem Move forward
C$=C$+"Let Y=Y+8 ; " : Rem Move Sprite down screen
C$=C$+"For R0=1 To 320 ; Let X=X-1 ; Next R0 ;" Rem Move back
C$=C$+"Let Y=Y+8 ; Next R1":Rem Move Sprite down
Amal 8,C$:Amal On 8
```

The above program generates a smooth but quite basic attack pattern. A further demonstration can be found in **Example 14.1** in the MANUAL folder.

Recording a complex movement sequence

PLay

PLay path

If you've looked at the smooth attack waves in a modern arcade game, and thought them forever beyond your reach, think again. The AMAL Play command allows you freely animate your objects through practically any sequence of movements you can imagine. It works by playing a previously defined movement pattern stored in the AMAL memory bank.

These patterns are created from the AMAL accessory on the AMOS program disc. This simply records a sequence of mouse movements and enters them directly into the AMAL memory bank. Once you've defined your patterns in this way, you can effortlessly assign them to any object on the screen, reproducing your original patterns perfectly. Both the speed and the direction of your movement can be changed at any time from your AMOS Basic program.

The first time AMAL encounters a play command, it checks the AMAL bank to find the recorded movement you specified using the *path* parameter. *path* is simply a number ranging from one to the maximum number of patterns in the bank. If a problem crops up during this phase, AMAL will abort the play instruction completely, and will skip to the next instruction in your animation string.

After the pattern has been initialised, register R0 will be loaded with the tempo of the movement. This determines the time interval between each individual movement step. All timings are measured in units of a 50th of a second. By changing this register within your AMAL program, you can speed up or slow down your object movements accordingly.

Note that each movement step is **added** to the current coordinates of your object. So if an object is subsequently moved using the Sprite or Bob instructions, it will continue it's manoeuvres unaffected, starting from the new screen position. It's therefore possible to animate dozens of different objects on the screen using a single sequence of movements.

Register R1 now contains the a flag which sets the direction of your movements. There are three possible situations:

- R1>0 Forward

A value of one for R1 specifies that the movement pattern will be replayed from start to finish, in exactly the order it was created. (Default)

- R1=0 Backward

Many animation sequences require your objects to move back and forth across the screen in a complex pattern. To change direction, simply load R1 with a zero. Your object will now turn around and execute your original movement steps in reverse.

- R1=-1 Exit

If a collision has been detected from your AMOS program, you'll need to stop your object completely, and generate an explosion effect. This can be accomplished by setting R1 to a value of minus one. AMAL will now abort the play instruction, and immediately jump to the next instruction in your animation sequence.

The clever thing about these registers is that they can be changed directly from AMOS Basic. This lets you control your movement patterns directly from within your main program. There's even a special AMPLAY instruction to make things easier for you.

The PLay command is perfect for controlling the aliens in an arcade game. In fact, it's the single most powerful instruction in AMAL.

AMAL *(Call an AMAL program)*

AMAL n,a$
AMAL n,p
AMAL n,a$ to address

The AMAL command assigns an AMAL program to an animation channel. This program can be taken either from a string in *a$* or directly from the AMAL bank.

The first version of the instruction loads your program from the string *a$* and assigns it to channel *n*. *a$* can contain any list of AMAL instructions. Alternatively you can load your program from a memory bank created using the AMAL accessory. *p* now refers to the number of an AMAL program stored in bank number 4.

n is the number of an animation channel ranging from 0 to 63. Each AMOS channel can be independently assigned to either a bob, a sprite or a screen.

Only the first 16 AMAL programs can be performed using interrupts. In order to exceed this limit you need execute your programs directly from Basic using the SYNCHRO command.

The final version of the AMAL instruction is provided for advanced users. Instead of moving an actual object, this simply copies the contents of registers X,Y and A into a specific area of memory. You can now use this information directly in your own Basic routines. It's therefore possible to exploit the AMAL system to animate anything from a BLOCK to a character. The format is:

AMAL n, a$ To address

address must be EVEN and must point to a safe region of memory, preferably in an AMOS string or a memory bank. Every time your AMAL program is executed (50 times per second), the following values will be written into this memory area:

Location	Effect
Address	Bit 0 is set to 1 If the X has changed. Bit 1 indicates that Y has been altered. Bit 2 will be set if the image (A) has changed since the last interrupt.
Address+2	Is a **word** containing the latest value of X
Address+4	Holds the current value of Y
Address+6	Stores the value of A.

These values can be accessed from your program using a simple DEEK.
Note: This option totally overrides any previous CHANNEL assignments.

AMAL Commands

Here is a full list of the available AMAL commands:

Move

This moves an object smoothly from one position to another. The syntax is:

Move deltaX, deltaY, steps

deltaX holds the distance to be moved horizontally. Positive numbers indicate a movement from left to right, and negative values from right to left.

deltaY specifies the vertical displacement. If *deltaY* is positive then your object will move down the screen, otherwise it will drift upwards.

n indicates the number of steps the movement is to be performed in. The smoothest movements are generated when both *deltaX* and *deltaY* are exact multiples of *n.*

A (Anim)

Anim cycles,(image,delay)(image,delay)...

The Anim instruction assigns a sequence of images to either a sprite or a blitter object to generate a realistic animation effect.

cycles specify the number of times the animation is to be repeated. If it's set to zero, the animation will continue indefinitely. *image* chooses the image number for each frame of your animation. *delay* sets the amount of time (in 50ths of a second) the image will be displayed.

After the Anim command has been initialised, AMAL will automatically jump to the next instruction. This allows you to combine both animation and movement in the same AMAL program.

Let

Let reg=exp

This command assigns a value to an AMAL register. *reg* is the name of the AMAL register to be changed. There are 10 internal registers ranging from R0 to R9 available for your use, and a further 26 external registers (RA to RZ). You can also alter the position and type of your object directly using the special registers X,Y and A.

expr is a standard arithmetical expression and is evaluated from left to right to produce the final result.

Most of the normal operators are supported including +,-,* and /. However you are not allowed to change the order of calculation using brackets "()".

Jump

Jump L

The Jump command jumps from the current point in your AMAL program to label *L*. *L* is the name of a label which has been previously defined in your AMAL string. Labels consist of single capital letter and are created using a ":" as in standard Basic.

If

If exp Jump L

The If instruction allows you to jump from one part of an AMAL program to another depending on the result of a test. *exp* is a logical expression in the standard format

If *exp* is TRUE then the program will jump to label *L*, otherwise it will immediately execute the next instruction after the Jump.

There are two other forms of this command which are used by the AUTOTEST feature:

If exp Direct L	(Chooses part of program to be executed after an autotest)
If exp eXit	(Leaves Autotest)

See AUTOTEST for more information.

For To Next

For Reg=start To end ...Next Reg

This is a direct implementation of Basic's FOR...NEXT loops. *Reg* can be any internal or external AMAL register. As normal, loops can be nested but the step size of your loop is always set to one.

Note that AMAL will automatically wait for the next vertical blank before jumping back to the start of your loop with Next. Since the object movements in your program will only be seen after the screen is updated after the VBL, faster loops would simply waste valuable processor time with no visible effect. So your For...Next loops are automatically synchronized with the screen updates to produce the smoothest possible results.

PLay

PLay path

The PL command animates your objects through a series of movements stored in the AMAL bank. These patterns are entered directly with the mouse, using the powerful AMAL accessory utility. So there's no real limit to the type of patterns you can produce with this system.

path is the number of a pattern which has been previously saved in the AMAL bank. If this pattern does not exist, AMAL will skip the PL instruction, and immediately jump to the next command in your animation sequence.

All movements are performed relative to the current position of your objects. It's therefore possible to move an entire attack wave using a single path definition. You can also move an object directly from Basic without affecting the movement in the slightest. The status of the current movement is controlled through two AMAL registers.

R0 holds the tempo of your movement. Increasing this value will speed up the object on the screen.

R1 contains the direction of the motion. There are three possible alternatives.

R1>0 Moves through the movement sequence in the original order .

R1=0 Executes your movement steps in reverse.

R1=-1 Stops the movement sequence completely and proceeds to the next AMAL instruction.

The contents of these registers can be changed at any time from within your Basic program using either the AMREG or the special AMPLAY command.

A further explanation of this instruction can be found in the AMAL tutorial near the beginning of this chapter. Also see **Example 14.2** in the MANUAL folder.

Warning: It is essential that you use semi-colons to split up your AMAL instructions. The following string will generate an *AMAL bank not reserved* error simply because there is no separator.

```
A$="Pause Let R0=1"
```

The correct syntax is:

```
A$="Pause ; Let R0=1"
```

End

End

Terminates the entire AMAL program and turns off the Autotest feature if it's been defined.

Pause

Pause

Pause temporarily halts the execution of your AMAL program and waits for the next vertical blank period. After the VBL your program will be automatically resumed starting from the next instruction.

Pause is often used before a Jump command to ensure that the number of jumps is less than the maximum of 10 per VBL. This frees valuable processor time for your Basic programs, and can have a dramatic effect on their overall speed. So try to get into the habit of preceding your Jump commands with a Pause instruction as it's much more efficient.

AUtotest

AU (List of tests)

The Autotest feature of AMAL has been designed to provide fast interaction between AMAL and the user. It adds a special test at the start of the AMAL program which is performed every VBL before the rest of the AMAL program is executed. See the Autotest system for more details.

e**X**it

eXit

Exits from an Autotest and re-enters the current AMAL program.

Wait

Wait

Wait freezes your AMAL program and only executes the Autotest.

On

On

ON activates the main program after a wait command.

Direct

Direct

Sets the section of the main program to be executed after an autotest.

AMAL functions

=XM *(Returns the X coordinate of the mouse)*

This function is exactly the same as the X MOUSE function in AMOS Basic. It returns the X coordinate of mouse cursor in hardware coordinates.

=YM *(Returns the Y coordinate of the mouse)*

YM returns the Y coordinate of the mouse pointer as a hardware coordinate.

=K1 *(Status of left mouse key)*

K1 returns a value of -1 (true) if the left mouse key has been pressed, otherwise 0 (false).

=K2 *(Status of right mouse key)*

Returns the state of the right mouse button. If the button has been pressed then K2 will return -1 (true).

=J0 *(Tests right joystick)*

The J0 function tests the right joystick and returns a bit-map containing the current status. See JOY for more details.

=J1 *(Test left joystick)*

This tests the left joystick and returns a bit-pattern in standard format.

=Z(n) *(Random number)*

The Z function returns a random number from -32767 to 32768. This number can be limited to a specific range using the bit-mask *n*.

A logical AND operation is performed between the bit mask *n* and the random number to generate the final result. So setting *n* to a value of 255 will ensure that the numbers will be returned in the range 0 to 255.

Since this function has been optimized for speed, the number returned isn't totally random. If you need really random numbers, you would be better to generate your values using Basic's RND and then load them into an external AMAL register with the AMREG function.

=XH *(Convert a screen x coordinate into a hardware coordinate)*

=XH(s,x)

This converts a screen *x* coordinate into its equivalent hardware coordinate relative to screen *s*.

=YH *(Converts a screen Y coordinate into hardware format)*

=YH(s,y)

YH transforms a *y* coordinate from screen format into hardware format relative to screen *s.*

=XS *(Hardware to screen conversion)*

=XS(s,x)

Changes hardware coordinate *x* into a graphic coordinate relative to screen *s.*

=YS *(Hardware to screen conversion)*

=YS(s,y)

Transform hardware *y* coordinate into its equivalent screen coordinate.

=BC *(Check for collisions between bobs)*

=Bob Col(n,s,e)

BC is identical to the equivalent AMOS Basic BOB COL instruction. It checks bob number *n* for collisions between bobs *s* to *e.*

If a collision has been detected, then BC will return a value of -1 (true), otherwise 0 (false). This instruction may **not** be performed within an interrupt. So it's only available when you are executing you AMAL routines directly from Basic with the SYNCHRO instruction.

=SC(m,s,e) *(Sprite Collisions)*

=Sprite Col(n,s,e)

This is equivalent to the SPRITE COL function in AMOS Basic. It checks sprite *n* for collisions between sprites *s* to *e.* If the test is successful, a value of -1 (true) will be returned. Like the previous BC function it is only allowed in conjunction with the SYNCHRO instruction.

=C(n) *(Col)*

Returns the status of object *n* after an SC or BC function. If the object has collided then this function will return a value of -1 (true), otherwise 0 (false).

=V(v) *(Vumeter)*

The VU function samples one of the sound channels and returns the intensity of the current voice. This is a number in the range 0-255. You can use this information to animate your objects in time to the music. An example of this can be found in **Example 14.3.** Also see the VUMETER function from AMOS Basic.

Controlling AMAL from Basic

AMAL ON/OFF *(Start/stop an AMAL program)*

AMAL ON [n]

Once you've defined your AMAL program you need to execute it using the AMAL ON command. This activates the AMAL system and starts your programs from the first instruction.

AMAL ON activates all your programs. The optional parameter *n* allows you to start just one routine at a time.

AMAL OFF [n]

Stops one or all AMAL programs from executing. These programs are erased from memory. They can only be restarted by redefining them again using the AMAL instruction.

AMAL FREEZE *(Temporarily freeze an AMAL program)*

AMAL Freeze [n]

Stops one or more AMAL programs from running. Your programs can be restarted at any time using a simple call to AMAL ON. Note that this instruction should always be used to stop AMAL before a command such as DIR is executed, otherwise problems with timing can cause visual mishaps.

=AMREG= *(Get the value of an external AMAL register)*

r=AMREG(n, [channel])
AMREG(n, [channel])=expression

The AMREG function allows you to access the contents of internal and external AMAL register directly from within your Basic program.

n is the number of the register. Possible values range from 0 to 25 with zero representing register RA and twenty-five denoting RZ.

By using the optional *channel* parameter you can reference any AMAL internal register. In this mode *n* ranges between 0 and 9 representing R0 to R9.

The following example shows how it is possible to retrieve a sprite's current X position from Basic:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Channel 1 To Sprite 8 : Sprite 8,100,100,1
A$="Loop: Let RX=X+1; Let X=RX; Pause; Jump Loop" : Rem X will overflow when >640
```

```
Amal 1,A$ : Amal On : Curs Off
Do
   Locate 0,0
   Z=Asc("X")-65 : Rem Note the use of ASC to get the register number
   Print Amreg(Asc("X")-65)
Loop
```

AMPLAY *(Control an animation produced with PLay)*

AMPLAY tempo,direction[start TO end)]

Any movement sequences you've produced using the AMAL PL command are controlled through the internal registers R0 and R1. Each object will be assigned it's own unique set of AMAL registers. So if you're animating several objects, you'll often need to load a number of these registers with exactly the same values.

Although this can be achieved using the standard AMREG function, it would obviously be much easier if there was a single instruction which allowed you to change R0 and R1 for a whole batch of objects at a time. That's the purpose of the AMPLAY command.

AMPLAY takes the *tempo* and *direction* of your movements, and loads them into the registers R0 and R1 in the selected channels.

tempo controls the speed of your object. on the screen. It sets a delay (in 50ths of a second) between each successive movement step.

direction changes the direction of the motion Here's a list of the various different options.

Value	Direction of motion
>0	Move the selected object in the original movement direction.
0	Reverses the motion and moves the object backwards.
-1	Aborts movement pattern and jumps to the following instruction in your AMAL animation sequence.

As a default, this instruction will affect all current animation channels. This can be changed by adding some explicit *start* and *end* points to the command. *start* is the channel number of the first object to be adjusted. *end* holds the channel number assigned to the last object in your list.

Note that either the *tempo* or the *direction* can be omitted as required. Examples:

```
Amplay ,0 : Rem Reverse your objects
Amplay 2, : Rem Slow down your movement patterns
Amplay 3,1 : Rem Set temp to three and direction to 1
Amplay ,-1 3 To 6 : Rem Stop movements on channels 3,4,5 and 6
```

=CHANAN *(Test Amal animation)*

s=CHANAN(channel)

This is a simple function which checks the status of an AMAL animation sequence and returns -1 (true) if it is currently active or 0 (false) if the animation is complete. *channel* holds the number of the channel to be tested. Here's an example:

```
Load "AMOS_DATA:Sprites/Monkey_right.abk" : Get Sprite Palette
Sprite 9,150,150,11
M$="Anim 12,(11,4)(12,4)(13,4)(14,4)(15,4)(16,4);"
Amal 9,M$ : Amal On
While Chanan(9)
Wend
Print "Animation complete"
```

=CHANMV *(Checks whether an object is still moving)*

s=CHANMV(channel)

Returns a value of -1 (true) if the object assigned to *channel* is currently moving, otherwise 0 (false).
This command can be used in conjunction with the AMAL Move instruction to check whether a movement sequence has "run out" of steps. You can now restart the sequence at the new position with an appropriate movement string if required. Example:

```
Load "AMOS_DATA:Sprites/Monkey_right.abk" : Get Sprite palette
Sprite 9,150,50,11
M$="Move 300,150,150; Move -300,-150,75"
Amal 9,M$ : Amal On
While Chanmv(9)
Wend
Print "Movement complete"
```

AMAL errors

=AMALERR *(Return the position of an error)*

p=AMALERR

AMALERR returns the position in the current animation string where an error has occurred. Careful inspection of this string will allow you to quickly correct your mistakes. Example:

```
Load "AMOS_DATA:Sprites/Octopus.abk"
Sprite 8,100,100,1
A$="L: IF X=300 then Jump L else X=X+1; Jump L"
Amal 8,A$
```

This program will generate a syntax error because IF will be interpreted as the two instructions "I" and "F". To find the position in the animation string of this error, type the following instruction from the direct window.

```
Print Mid$(A$,Amalerr,Amaller+5)
```

Error messages

If you make a mistake in one of your AMAL programs, AMOS will exit back to Basic with an appropriate error message. Here's a full list of the errors which can be generated by this system, along with an explanation of their most likely causes.

Bank not reserved: This error is caused if you attempt to call the PLay instruction without first loading a bank containing the movement data into memory. This should be created with the AMAL accessory program. If you not using PLay at all then check that you've correctly separated any Pause and **L**et instructions in your program.

Instruction only valid in autotest: You've inadvertently called either the Direct or the eXit instructions from your main AMAL program.

Illegal instruction in Autotest: Autotest may only be used in conjunction with a limited range of AMAL commands. It's not possible to move or animate your objects in any way inside an autotest. So check for erroneous commands like Move, Anim or For..Next.

Jump To/Within Autotest in animation string: The commands inside an autoest function are completely separate from your main AMAL program. So AMAL does not allow you to jump directly inside an AUtotest procedure. To leave an autoest,and return to your main AMAL program you must use either eXit or Direct .

Label already defined in animation string: You've attempted to define the same label twice in your AMAL program. All AMAL labels consist of just a single CAPITAL letter. So **Test** and **Total** just different versions of the same label (**T**). This error is also generated if you have accidentally separated two instructions by a ":" (colon). Use a semi-colon instead.

Label not defined in animation string: This error is generated when you try to jump to a label which does not currently exist in your animation string.

Next without For in animation string: Like it's Basic equivalent each For command should be matched by a corresponding Next. statement. Check any nested loops for an spurious Next command.

Syntax error in animation string: You've made a typing mistake in one of your animation strings. It's easy to cause this error by accidentally entering an AMAL instruction in full, just like it's Basic equivalent.. Remember that AMAL commands only consist of one or letters CAPITALS. So If you attempt to type instructions like:FOR or NEXT you'll get an error. The correct syntax of these commands are For..Next

Animation channels

AMOS allows you to execute up to 64 different AMAL programs simultaneously. Each program is assigned to a specific animation *channel.*

Only the first 16 channels can be performed using interrupts. If you need to animate more objects you'll have to turn off the interrupts using SYNCHRO OFF. You can now execute the AMAL programs step by step using an explicit call to the SYNCHRO command in your main program loop. As a default, all interrupt channels are assigned to the relevant hardware sprite.

CHANNEL *(Assign an object to an AMAL channel)*

CHANNEL n TO object s

The CHANNEL command assigns an animation channel to a particular screen related *object.* In AMAL, you're not restricted to a single channel per object. however. Any single screen object can be safely animated with several channels if required. There are various different forms of this instruction.

Animating a computed sprite

CHANNEL n TO SPRITE s

This assigns sprite number *s* to channel *n.* As a default, channels from 0 to 7 are automatically allocated to the equivalent hardware sprite, and 8 to 15 are reserved for the appropriate computed sprites.

In order to animate the computed sprites from 16 onwards, you'll need to allocate them directly to an animation channel with the CHANNEL command. As normal, sprite numbers from 8 to 63 specify a computed sprite rather than a single hardware sprite. For example:

```
Channel 5 To Sprite 8:Rem Animates computed sprite 8 using channel 5.
```

The X,Y registers in your AMAL program now refer to the hardware coordinates of the selected sprite. Similarly the current sprite image is held in register A.

Animating a blitter object

AMAL programs can also be used to animate blitter objects.

CHANNEL n TO BOB b

Allocates blitter object *b* to animation channel *n.* This object will be treated in an identical way to the equivalent hardware sprite. The only difference is that registers X and Y now contain the position of your bob in **screen** coordinates.

Note that if you've activated screen switching with the DOUBLE BUFFER command, this will be automatically used for all bob animations. For a complete example see **8** from the MANUAL folder.

Moving a screen

AMOS Basic allows you to freely position the current screen anywhere on your TV display. Normally this is controlled with the SCREEN DISPLAY instruction. However, sometimes it's useful to be able to move the screen using interrupts.

CHANNEL n TO SCREEN DISPLAY d

This sets the channel *n* to screen number *d*. Screen *d* can be defined anywhere in your program. You'll only get an error if the screen hasn't been opened when you start your animation.

The X and Y variables in AMAL now hold the position of your screen in hardware coordinates. Register A is **not** used by this option and you can't animate screens using Anim. Otherwise all standard AMAL instructions can be performed as normal. So you can easily use this system to "bounce" the picture around the display.Examples:

```
Load Iff "AMOS_DATA : IFF/Frog_screen.IFF",1
Channel 0 To screen display 1
Amal 0,"Loop: Move 0,200,100 ; Move 0,-200,100 ; Jump Loop"
Amal on 0 : Direct

Load Iff "AMOS_DATA : IFF/Frog_screen.IFF",1
Channel 0 to screen display 1
Rem Screen can only be displayed at certain postions in the X
Amal 0,"Loop: Let X=XM; Let Y=YM; Pause; Jump Loop"
Amal On : Direct
```

For a further example of this technique, load **Example 14.4** from the MANUAL folder. This demonstrates how the SCREEN DISPLAY can be used in conjunction with the menu commands to slide the menu screen up and down your display. It's similar to the display system found in Magnetic Scrolls' excellent series of adventures.

Hardware Scrolling

Although hardware scrolling can be performed using AMOS Basic's SCREEN OFFSET command, it's often easiest to animate your screens using AMAL instead as this generates a much smoother effect.

CHANNEL n TO SCREEN OFFSET d

This assigns AMAL program number *n* to a screen *d*, for the purpose of hardware scrolling. The X and Y registers now refer to the section of the screen which is to be displayed through your TV. Changing these registers will scroll the visible screen area around the display. Here's an example:

```
Screen Open 0,320,500,32,lowres : Rem Open an extra tall screen
Screen Display 0,,45,320,250     AMOSPIC.IFF
Load Iff "AMOS_DATA:IFF/~~Magic_Screen.IFF~~"
Screen Copy 0,0,0,320,250 To 0,0,251
Screen 0 : Flash off : Get palette (0)
Channel 0 to Screen Offset 0
Amal 0,"Loop: Let X=XM-128; Let Y=YM-45; Pause; Jump Loop"
Amal On : Wait Key
```

This program allows you to scroll through the screen using the mouse. Try moving the mouse in direct mode. For a further example of hardware scrolling, see **Example 14.5**.

Changing the screen size

CHANNEL n TO SCREEN SIZE s.

This allows you to change the size of a screen using AMAL. *s* is the number of the screen to be manipulated. Registers X and Y now control the width and height of your screen respectively. They're similar to the W and H parameters used by the SCREEN DISPLAY command. Example:

```
Load Iff "AMOS_DATA:IFF/Magic_Screen.IFF",0
Channel 0 To Screen Size 0
Screen Display 0,,,320,1 : Rem Set the screen size to 1
A$="Loop: For R0=0 To 255 ; Let Y=R0 ; Next R0; "
A$=A$+"For R0=0 To 254; Let Y=255-R0; Next R0; J Loop"
Amal 0,A$ : Amal On : Direct
```

Rainbows

CHANNEL n TO RAINBOW r

This option generates a rainbow effect within an AMAL program. As usual *n* is the number of an animation channel from 0 to 63. *r* is an identification number of your rainbow (0-3).

X holds the current BASE of your rainbow. This is the first colour of your rainbow palette to be displayed. Changing it will make the rainbow appear to turn. Y contains the line on the screen at which the rainbow effect will start. If you alter this value, the rainbow effect will move up or down. All coordinates are measured in **hardware** format.

Register A stores the height of your rainbow on the screen. A demonstration of this system can be found in **11.** See the AMOS Basic RAINBOW command for more details.

Advanced Techniques

The AUTOTEST system

Normally all AMAL programs are performed in strict order from start to finish. Inevitably some commands such as Move and For...Next will take several seconds to complete. Although this will be fine in the vast majority of cases it may lead to significant delays in the running of certain programs. Take the following simple program:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
```

```
Sprite 8,130,50,1
Amal 8,"Loop: Let R0=XM-X; Let R1=YM-Y; Move R0,R1,50; Jump Loop"
Amal On : Direct
```

As you move the mouse, the sprite is supposed to follow it around on the screen. However in practice the response time is quite sluggish, because the new values of XM and YM are only entered after the sprite movement has totally finished. Try moving the mouse in a circle. The octopus is completely fooled!

Autotest solves this problem by performing your tests at the start of every VBL, before continuing with the current program. Your tests now occur at regular 1/50 intervals, leading to a practically instantaneous response!

Autotest commands

The syntax of Autotest is:

AUtotest (tests)

tests can consist of any of the following AMAL commands.

Let

L reg=exp

This is the standard AMAL Let instruction. It assigns the result of an expression to register *reg.*

Jump

Jump label

The Jump command jumps to another part of the current autotest. *Label* is defined using the colon ":" and **must** lie inside the autotest brackets.

e**X**it

Leaves the autotest and re-enters the main program from the point it left off.

Wait

Wait turns off the main AMAL program completely, and only executes the Autotest.

If

In order to simplify the testing process inside an autotest routine there's a specially extended version of the AMAL If statement. This allows you to perform one of three actions depending on the result of the logical expression *exp.*

if exp Jump L	(Jumps to another part of the autotest)
If exp Direct L	(Chooses part of the program to be executed after an autotest)
If exp eXit	(Leaves autotest)

On

ON restarts the main program again after a previous Wait instruction. This lets you wait for a specific event such as a mouse click without wasting valuable processor time.

Direct

Direct label

Direct changes the point at which the main program will be resumed after your test. AMAL will now jump to this point automatically at the next vertical blank period. Note that *label* **must** be defined outside the Autotest brackets.

Inside Autotest

Here's the previous example rewritten using the Autotest feature.

```
Load "AMOS_DATA:Sprites/octopus.abk"
Sprite 8,130,50,1 : Get Sprite Palette
A$="AUtotest (If R0<>XM Jump Update"
A$=A$+"If R1<>YM Jump Update else eXit"
A$=A$+"Update: Let R0=XM; Let R1=YM; Direct M)" : Rem End of autotest
A$=A$+"M: Move R0-X,R1-Y,20 Wait;" : Rem Try changing 20 to different values!
Amal 8,A$ : Amal on
```

The sprite now smoothly follows your mouse, no matter how fast you move it. The action of this program is as follows:

Every 50th of a second the mouse coordinates are tested using the XM and YM functions. If they are unchanged since the last test, the Autotest is aborted using the eXit command. The main program now resumes precisely where it left off.

However if the mouse has been moved, the autotest routine will restart the main program again from the beginning (label **M**) using the new coordinates in XM and YM respectively.

Timing considerations

UPDATE EVERY *(Save some time for your Basic programs)*

UPDATE EVERY n

Although most AMAL programs are performed practically instantaneously, any objects they manipulate need to be explicitly drawn on the Amiga's screen.

The amount of time required for this updating procedure is unpredictable and can vary during the course or your program. This can lead to an annoying jitter in the movement patterns of certain objects.

The UPDATE EVERY command slows down the updating process so that even the largest object can be redrawn during a single screen update. This regulates the animation system and generates delightfully smooth movement effects.

n is the number of vertical blank periods (50ths of second) between each screen update. In practice you should start off with a value of two, and gradually increase it until movement is smooth.

One useful side effect of UPDATE EVERY, is to reserve more time for Basic to execute your programs. With judicious use of this instruction, it's sometimes possible to speed up your programs by as much as 30%, without destroying the smoothness of your animation sequences.

Beating the 16 object limit

SYNCHRO *(Execute an AMAL program directly)*

SYNCHRO [ON/OFF]

Normally AMOS Basic will allow you to execute up to 16 different AMAL programs at a time. This limit is determined by the overall speed of the Amiga's hardware. Each AMAL program takes its own slice of the available processor time. So if you're using the standard interrupt system, there's only enough time to execute around 16 separate programs.

The SYNCHRO command allows you to exceed this restriction by executing your AMAL programs directly from Basic. Instead of using interrupts, all AMAL programs are now run using a single call to the SYNCHRO command. Since AMAL programs execute far faster than the equivalent Basic routines, your animations will still be delightfully smooth. But you will now be able to decide when and where your AMAL routines will be performed in your program.

One additional bonus is that you can now include collision detection commands such as Bob Col or Sprite Col directly in your AMAL routines. These are not available from the interrupt system as they make use of the Amiga's blitter chip. This would be impossible using interrupts.

Before calling SYNCHRO you first need to turn off the interrupts with SYNCHRO OFF. It's important to do this **before** defining your AMAL programs, otherwise you won't be allowed to use channel numbers greater than 15 without an error.

Due of the sheer power of the animation system, it's nearly possible to write entire arcade games completely in AMAL. This leaves your Basic program with simple jobs such as managing the hi-score table and loading your attack waves from the disc. The results will be indistinguishable from pure machine code. A good example is Cartoon Capers, the first commercial games release that's written entirely in AMOS.

A demonstration of SYNCHRO can be found in **Example 14.6** from the MANUAL folder.

STOS compatible animation commands

The original STOS Basic included a powerful animation system which allowed you to move your sprites in quite complex patterns using interrupts. At the time, these commands were hailed as a breakthrough.

Although they've now been overshadowed by the AMAL system, they do provide a simple introduction to animation on the Amiga. So AMOS provides you with the entire STOS animation system as an extra bonus!

If you're intending to convert STOS programs to AMOS, you'll need to note the following points:

- Unlike STOS, the movement patterns in AMOS Basic can be assigned to any animation channel you like. The Move commands can therefore be used to move bobs, sprites or screens, using exactly the same techniques.

  As a default, all animation channels are assigned to the equivalent hardware sprites. In practice you may find it easier to substitute blitter objects as these are much closer to the standard STOS Basic sprites. Add a sequence of CHANNEL commands to the start of your program like so:

  ```
  Channel 1 to bob 1
  Channel 2 to bob 2
  :      :      :
  ```

  Don't forget to call DOUBLE BUFFER during your initialisation procedure, otherwise your bobs will flicker annoyingly when they are moved.

- The same channel can be used for both STOS animations and AMAL programs. So it's easy to extend your programs once they've been successfully converted into AMOS Basic. The order of execution is:

  AMAL
  MOVE X
  MOVE Y
  ANIM

MOVE X *(Move a sprite horizontally)*

```
MOVE X n,m$
```

MOVE X defines a list of horizontal movements which will be subsequently performed on animation channel number *n*.

n can range from 0 to 15 and refers to an object you have previously assigned using the CHANNEL command. *m$* contains a sequence of instructions which together determine both the speed and direction of your object. These commands are enclosed between brackets and are entered using the following format:

```
(speed,step,count)
```

There's no limit to the number commands you can include in a single movement string, other than the amount of available memory.

speed sets a delay in 50ths of a second between each successive movement step. The speed can vary from 1 (very fast) to 32767 (incredibly slow).

step specifies the number of pixels the object will be moved during each operation.

If the step is positive the sprite will move to the right, and if it is negative it will move left.

The apparent speed of the object depends on a combination of the speed and step size. Large displacements coupled with a moderate speed will move the object quickly but jerkily across the screen. Similarly a small step size combined with a high speed will also move the object rapidly, but the motion will be much smoother. The fastest speeds can be obtained with a displacements of about 10 (or -10).

count determines the number of times the movement will be repeated. Possible values range from 0 to 32767. A *count* of 0 performs the movement pattern indefinitely.

In addition to the above commands, you can also add one of the following directives at the end of your movement string.

The most important of these extensions is the L instruction (for loop), which jumps back to the start of the string and reruns the entire sequence again from the beginning. Example:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Sprite 1,130,100,1 : Rem Define Sprite 5
Move X 1,"(1,5,60)(1,-5,60)L"
Move On
```

The E option allows you to stop your object when it reaches a specific point on the screen. Change the second to last line in the above example to:

```
Move X 1,"(1,5,30)E100"
```

Note that these end-points will only work if the x coordinate of the object exactly reaches the value you originally designated in the instruction. If this increment is badly chosen the object will leap past the end-point in a single bound, and the test will fail. Example:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Channel 1 To Sprite 8 : Channel 2 To Sprite 10
Print At(0,5)+"Looping OK"
Sprite 8,130,100,1
Move X 1,"(1,10,30)(1,-10,30)L"
Move On
Print At(0,10)+"Now press a key" : Wait Key
Sprite 10,140,150,2
Move X 2,"(1,15,20)L" : Move On 2
Print At(0,15)+"Oh dear!" :Wait Key
```

MOVE Y *(Move an object vertically)*

MOVE Y n,m$

This instruction complements the MOVE X command by enabling you to move an object vertically along the screen. As before, *n* refers to the number of an animation sequence you've allocated using the CHANNEL command, and ranges between 0 and 15.

m$ holds a movement string in an identical format to MOVE X. Positive displacements now correspond to a downward motion, and negative values result in an upward movement. Examples:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Channel 1 To Sprite 8 : Sprite 8,130,10,1 : Rem Install sprite
Move Y 1,"10(1,1,180)L" : Rem Loop sprite from 10,10 to 190,10 continually
Channel 2 To Screen Display 0 : Rem Assign the screen postion to channel 2
Move Y 2,"(1,4,25)(1,-4,25)" : Rem Moves screen up and down
Move On : Wait Key
```

MOVE ON/OFF *(Start/stop movements)*

MOVE ON/OFF [n]

Before your movement patterns will be executed they need to be activated using the MOVE ON command.

n refers to the animation sequence you wish to start, and can range from 0 to 15. If it is omitted then all your movements will be activated simultaneously.

MOVE OFF has exactly the opposite effect: It stops the relevant movement sequences in their tracks.

MOVE FREEZE *(Temporarily suspend sprite movements)*

MOVE FREEZE [n]

The MOVE FREEZE command temporarily halts the movements of one or more objects on the screen. These objects can be restarted again using MOVE ON.

n is completely optional and specifies the number of a single object to be suspended by this instruction.

=MOVON *(Return movement status)*

x=MOVON(n)

MOVON checks whether a particular object is currently being moved by the MOVE X and MOVE Y instructions. It returns -1 (true) if object *n* is in motion, and 0 (false), if it is stationary. Do not confuse this with the MOVE ON command.

Note that MOVON only searches for movement patterns generated using the MOVE commands. It will not detect any animations generated by AMAL.

ANIM *(Animate an object)*

ANIM n,a$

Anim automatically flicks an object through a sequence of images creating a smooth animation effect on the screen. These animations are performed 50 times a second using interrupts, so they can be executed simultaneously with your Basic programs.

n is the number of the channel which specifies a sprite or bob to be animated by this instruction.

a$ contains a series of instructions which define your animation sequence. Each operation is split into two separate components enclosed between round brackets.

image is number of the image to be displayed during each frame of the animation. *delay* specifies the length of time this image will be held on the screen (in 50ths of a second). Example:

```
Load "AMOS_DATA:Sprites/octopus.abk" : Get Sprite Palette
Channel 1 To Sprite 8 : Sprite 8, 200,100,1
Anim 1,"(1,10)(2,10)(3,10)(4,10)"
Anim on : Wait key
```

Just as with the MOVE instruction, there's also an L directive which enables you to repeat your animations continuously. So just change the ANIM command in the previous example to the following:

```
Anim 1,"(1,10)(2,10)(3,10(4,10)L"
```

ANIM ON/OFF *(Start an animation)*

ANIM ON/OFF[n]

ANIM ON activates a series of animations which have been previously created using the ANIM command. *n* specifies the number of an individual animation sequence to be initialised. If it is omitted then all current animation sequences will be started immediately.

ANIM OFF [n]

This halts one or more animation sequences started by ANIM ON.

ANIM FREEZE *(Freeze an animation)*

ANIM FREEZE [n]

ANIM FREEZE temporarily freezes the current animation sequence on the screen. *n* chooses a single animation sequence to be suspended. If it's not included, all current animations will be affected. They can be restarted at any time with a simple call to the ANIM ON instruction.

15:Background graphics

Nowadays, it's not uncommon for an arcade game to contain hundreds of different screens. With compaction, it's possible to cram a single 32 colour screen into about 30k of memory. So 100 screens would be the equivalent of about 3 megabytes of data. Imagine how difficult this would be to fit into a standard A500!

The classic way of avoiding this restriction, is to construct your backgrounds out of a set of simple building blocks. Once these *tiles* have been created, they can be placed on the screen in any order you like. So the same set of tiles can be reused to generate a vast number of potential screens. Each screen is now stored as a simple list of its components, and requires a tiny fraction of the original memory.

In order to exploit this system, you'll obviously need some way of defining your various screen maps. As you might have guessed, we've helpfully provided you with a powerful map definer accessory on the AMOS program disc. Full details can be found in the accompanying documentation file.

AMOS Basic also includes a number of special instructions for drawing your tiles on the screen. These make it easy to generate the fast scrolling backgrounds that are the hallmark of a modern arcade game.

Icons

Icons are separate images which have been especially designed for producing your background screens. Once you've drawn an icon, it's fixed permanently into place. So you can't move it to a new position using the AMAL animation system.

All icons are stored in their own AMOS memory bank (bank 2). This bank is created using the Sprite definer accessory (on the AMOS program disc), and will be automatically saved along with your Basic programs.

LIke Bobs, Icons are displayed using the Amiga's amazing Blitter chip. But since Icons are essentially static objects, they are usually drawn in REPLACE mode. Your icons will therefore totally erase any existing graphics at the current screen position.

PASTE ICON *(Draw an icon)*

PASTE ICON x,y,n

Draws icon number *n* on the screen at GRAPHIC coordinates *x,y*. *n* is the number of the icon which is to be displayed. This must have been previously stored in the ICON bank (2).

Icons can be freely positioned anywhere on the screen, subject to the normal clipping rules. Example:

```
Load "AMOS_DATA : Icons/Map_icons.abk"
Screen Open 0,320,256,32,Lowres : Cls 0 : Get Icon Palette
Rem Draw Border around screen
For X=1 To 11 : Paste Icon X*32,0,1 : Next X
For Y=1 To 6 : Paste Icon 0,Y*32+11 : Paste Icon 288,Y*32,1 : Next Y
For X=1 To 11 : Paste Icon X*32,223,1 : Next X
```

Note that if you're using double buffering, a copy of your icons will be drawn into both the physical and logical screens. Since this is rather slow, it's common practice to add a call to AUTOBACK 0 before drawing your icons on the screen. This restricts your icons to the current logical screen. You can then copy the entire background straight to the physical screen using SCREEN COPY, saving a considerable amount of time.

For a further example, see the MAPVIEW program on the AMOS DATA disc. This displays a background screen you've created using the AMOS Map Editor.

GET ICON *(Create an icon)*

GET ICON [s,] i,tx,ty TO bx,by

Captures an image from the screen and loads it into icon *i*. If this icon does not presently exist, it will be created for you in bank 2. This bank will be automatically reserved by the system if required.

i is the number of your icon, starting from 1. *tx,ty* to *bx,by* define the top and bottom corners of a rectangular zone which encloses the selected region.

s determines the number of the screen which will be used as the source of your image. If it's omitted, the image will be taken from the current screen instead. Example:

```
Erase 2
F$=Fsel$(“*.*”,””,”Load a screen”) : If F$=”” Then Direct
If Exist(f$) Then Load Iff f$,0 Else Direct
SH=Screen Height : H=SH/32-1 : Sw=Screen Width : W=SW/32-1
For Y=0 To H
  For X=0 To W
     Rem Grab an icon
     Get Icon X+Y*W+1,X*32,Y*32 To X*32+31,Y*32+31
  Next X
Next Y
Cls 0
Do
  Paste Icon Rnd(Sw-1),Rnd(SH-1),Rnd(H*W)+1
Loop
```

GET ICON PALETTE *(Get icon colours)*

GET ICON PALETTE

Grabs the colours of the icon images in bank 2, and loads them into the current screen palette. This command is normally used to initialize the screen after you've loaded some icons from the disc. Example:

```
Rem Load some icons from the AMOS DATA disc
Load “AMOS_DATA:Icons/Map_icons.abk”
Get Icon Palette
Paste Icon 100,100,1
```

DEL ICON *(Deletes icons)*

DEL ICON n[TO m]

Deletes one or more icons from the icon bank. *n* is the number of the first icon to be removed.

m is the optional number of the last icon to be deleted in the list. If it's included all the icons from *first* to *last* will be erased one after another. Example:

```
Load "AMOS_DATA:Icons/Map_icons.abk"
Paste Icon 100,100,1
Del Icon 1
Paste Icon 100,100,1
```

When the final icon in a bank has been deleted, the entire bank will be removed from memory.

MAKE ICON MASK *(Set colour zero to transparent)*

MAKE ICON MASK [n]

Normally, any icons you draw on the screen will completely replace the existing background. The icon will seem to be displayed in a rectangular box filled with colour zero.

If you want to avoid this effect and overlay your icons directly over the current graphics, you'll need to create a *mask* for your icons. This informs AMOS that colour zero should be treated as transparent.

n is the number of the icon to be affected. If it's omitted, a mask will be defined for all icons in the bank.

A demonstration of this effect can be seen in **EXAMPLE 15.1** in the MANUAL folder.

Screen blocks

AMOS Basic supplies you with a set of powerful BLOCK commands which allow you to grab part of an image into memory and paste it anywhere on the screen.

These instructions are mainly used for holding temporary data, since your blocks cannot be saved along with your Basic programs.

Blocks are especially effective in the construction of dialogue boxes, as they can be used to save the background areas before displaying your new graphics.

They can also be exploited in puzzle games like Split Personalities. Each block can be loaded with a single section of your image. You can then jumble your pictures by rearranging the blocks on the screen with PUT BLOCK.

GET BLOCK *(Grab a screen block into memory)*

GET BLOCK n,tx,ty,w,h[,mask]

GET BLOCK grabs a rectangular area in block number *n*, starting at coordinates *tx,ty.*

n is the number of the block ranging from 1-65535. *tx,ty* set the coordinates of the top left hand corner of your block. *w,h* hold the width and height of your block respectively.

mask is a flag which chooses whether a mask will be created for your new block.

mask=0	Replace mode. When the block is drawn on the screen, it will totally destroy any graphics at that current position.
mask=1	Calculates a mask for the block. Colour zero will now be treated as if it were transparent.

PUT BLOCK *(Copies a previously created block onto the screen)*

PUT BLOCK n[,x,y]
PUT BLOCK n,x,y,planes[,minterms]

PUT BLOCK copies block number *n* to the current screen. *x,y* specify the position of your new block on the screen. If they are omitted the block will be redrawn at its original screen coordinates.

Note that all drawing operations will be clipped to fit into the current screen, starting from the nearest 16 pixel boundary.

For a demonstration of the BLOCK commands see the routine in **EXAMPLE 15.2**. We've also provided experienced programmers with a couple of optional extras. These are not needed for the vast majority of applications, they're only required when you want to achieve weird special effects on the screen!

planes holds a bit-map which sets the range of colours which will be drawn in your block. The Amiga's screen is divided up into segments known as bit-planes.

Each plane contains a single bit for every point on the Amiga's screen. When the Amiga's hardware displays this point, it combines the bits from each plane to calculate the required colour number. Each bit in *planes* represents the status of a single bit-plane. If it's set to one, then the selected plane will be drawn by the instruction, otherwise it will be completely ignored. The first plane is represented by bit zero, the second by bit one, etc.

Usually, the block will be displayed in all the available bit-planes. This corresponds to a bit-pattern of %111111.

minterm selects the blitter mode used to copy your block on the screen. A full description of the possible drawing modes can be found in the section on SCREEN COPY.

The best way to learn about these options is to experiment!

DEL BLOCK *(Delete a screen block)*

DEL BLOCK n

Deletes one or more blocks and restores the memory used to AMOS Basic.

DEL BLOCK	Erases **all** current blocks
DEL BLOCK n	Deletes block number *n*.

GET CBLOCK *(Save and compact a screen image)*

GET CBLOCK n,x,y,sx,sy

The GET CBLOCK command saves and compacts a rectangular area of the screen. The compaction system used by this command has been especially optimized for speed. So it's nowhere near as efficient as the dedicated AMOS compression routines provided by the PACK or SPACK instructions.

CBLOCKS are often used to grab the area underneath your dialogue boxes. After the dialogue has completed, the screen can be quickly restored back to its original state. See **EXAMPLE 15.3** in the manual folder for a demonstration.

n specifies the number of your block and can range between 1 and 65535.

x/y are the coordinates of the block's top left corner. The *x* coordinate is rounded to the nearest multiple of 8.

w/h hold the dimensions of the area to be saved. The width of your block in *w* is always rounded to an exact multiple of eight.

PUT CBLOCK *(Displays a block created using CBLOCK)*

PUT CBLOCK n [,x,y]

Places block *n* on the current screen at coordinates *x,y*. If the target coordinates are omitted, the block will be redrawn at its original screen position. Also note that *x* is automatically rounded to the nearest eight pixel boundary.

DEL CBLOCK *(Deletes a screen block defined with GET CBLOCK)*

DEL CBLOCK [n]

Erases all blocks from memory. If *n* is present only block *n* will be deleted.

16: Menus

If you've used the Amiga for some time you'll already be familiar with the idea of menus. Impossible as it seems, AMOS has taken the existing system and improved it almost beyond recognition.

Menus can be created with up to eight separate levels, and each individual menu item can be repositioned on the screen at will. Menu titles can be printed in any combination of colours or styles. You can also include bobs or icons directly in your menus using an amazing menu definition language.

AMOS Basic is equally impressive when it comes to reading a menu. There's a built-in interrupt-driven ON MENU command which can automatically branch to a selected point in your program depending on the option selected. Furthermore, any menu option can be accessed directly from the keyboard using the MENU KEY instruction.

For a demonstration of the terrific effects that can be achieved with this system, load the program **EXAMPLE 16.1** which can be found in the Manual folder. You won't believe the sheer power of these commands until you've seen them in action!

Using a menu

All AMOS menus are called up by holding down the right mouse button in the standard way. Once a menu has been activated you can then select an option directly with the mouse cursor. When you release the button, the option number you have chosen will be returned to your program.

Menus can be repositioned by placing the mouse cursor over the top left corner of an item and holding down the **left** button. A small box will now appear on the menu bar which can be dragged across the screen using the mouse.

In addition, holding down the SHIFT key will freeze a menu into place. This allows you explore a menu without selecting any of the various options. You can also use any of the mouse features such as slowing or axis selection in conjunction with your menus.

Creating a simple menu

AMOS menus can be created either directly within your programs or using a special menu definer included on the AMOS program disc.

If you've never used menus before, the sheer variety of the available menu commands may seem a little overwhelming. Here's a brief description of some of the basic features to provide you with a painless introduction to AMOS menus.

Setting the title line

The first stage in the creation of a menu is to define the *title line*. The title line of a menu can be set using the MENU$ command. In its simplest form this has the format:

MENU$ *(Set a menu title)*

MENU$(n)=title$

MENU$ creates a title line for your menu. Each heading is assigned it's own individual number starting from one, and increasing from left to right. So the leftmost title is represented by a one, the next tile as a two, etc.

The text in *title$* holds the name of the option which will be displayed in your new menu. Here is a simple example which constructs a menu line consisting of just two titles: ACTION and MOUSE.

```
Menu$ (1)=” Action “
Menu$ (2)=” Mouse ”
```

Note the space after *Action* – this will separate it from *Mouse*, the next menu along. You must now specify a list of options to be associated with each of your new headings. These form a vertical bar which will drop into place whenever a title is selected with the mouse.

MENU$(t,o) *(Set a menu option)*

MENU$(t,o)=option$

This second form of MENU$ defines a set of options which will be displayed in the menu bar.

t is the number of menu heading which your option will be displayed under. *o* is the option number you wish to install in the menu bar. All options are numbered downwards from the top of the menu, starting from one.

The only physical limit to the size of your menu is the amount of memory, but it's wise to restrict yourself to less then about ten options for each title. This will keep the complexity of your menus down to an agreeable minimum.

option$ holds the name of your new option. This can consist of any section of text you like. For an example, try adding the following lines to the program above:

```
Rem Action menu
Menu$ (1,1)=” Quit ”
Rem Mouse menu
Menu$ (2,1)=” Arrow ”
Menu$ (2,2)=” Pointer ”
Menu$ (2,3)=” Clock ”
Wait key
```

This specifies a list of alternatives for the ACTION and the MOUSE menus. If you try to run this program as it stands, nothing will happen. That's because the menus need to be initialised with a call to the MENU ON command. Enter this in the above program before the "Wait Key" instruction. Now run the example and select the menu items with the mouse cursor. Remember to hold down the RIGHT mouse button first!

MENU ON *(Activate menu)*

MENU ON

MENU ON activates a menu defined using the MENU$ command. The menu line will now appear automatically when the RIGHT mouse button is pressed by the user. To start the previous menu, insert the following line after the definition statements.

```
Menu On
```

Go to the Direct window and play around with the menus. Select options by pressing the right mouse button.

Reading a simple menu

Once you've created your menu and activated the AMOS menuing system you'll want to discover which options have been selected by the user. This can be accomplished using a simple form of the CHOICE command.

=CHOICE *(Read a menu)*

selected=CHOICE

CHOICE returns a value of -1 (true) if the menu has been highlighted by the user, otherwise 0 (false). It's automatically reset to 0 (false) after each test. It's also possible to find the title number which has been selected using a second form of this instruction.

heading=CHOICE(1)

heading now contains the number of the *title* which has been highlighted by the user. Similarly you can retrieve the actual option number which has been chosen with a parameter of two.

item=CHOICE(2)

Try adding the following lines to the previous example:

```
Do
   Rem If Choice=-1 can be simplified to: If Choice , as seen below:
   If Choice and Choice(1)=1 Then Exit
   If Choice(1)=2 and Choice(2)<>0 Then Change Mouse Choice(2)
Loop
```

This changes the shape of the mouse cursor depending on which option you have chosen from the menu. A full demonstration of these menus can be found in the file **EXAMPLE 16.2** in the Manual folder.

Advanced menuing features

We will now cover some of the more advanced menuing features available from within AMOS Basic. Used properly these AMOS menus can add a whole new dimension to your programs.

MENU$ *(Create a menu)*

```
MENU$(,,)=normal$[,selected$][,inactive$][,background$]
```

MENU$ defines the appearance of each individual item in one of your menus. Unlike normal Amiga menus these items are not restricted to standard text. They can also include *embedded* commands which allow you to draw bobs, icons or graphics at any point in the menu line.

Any of the parameters in this instruction may be optionally omitted, so you can change parts of a menu description independently. A value of "" in your menu string will **erase** the existing setting. Similarly you can retain the original value by including a comma at the appropriate point. For example:

```
Menu$(1)=" Action ","" : Rem Erase second option
Menu$(2)=" Mouse 2 ",, : Rem Change title without altering anything else
```

The position of the menu item within the actual menu is indicated using a list of up to eight parameters separated by commas. The general format is:

```
(item)/(item, option)/(item, option, sub option)...
```

normal$* is a string which sets the normal appearance of an item when it is displayed in the menu. *selected$ changes the effect of highlighting a menu option with the mouse. As a default, selected items are printed in inverse text.

Inactive$ changes the appearance of an item which has been deactivated using the MENU INACTIVE command. If this string is omitted, all inactive items will be displayed in *italics.*

background$ creates a background for your menu items when they are initially drawn. Generally this will be a box of some sort created with the internal Bar or line commands.

From now on, we'll abbreviate these parameters using a standard notation:

```
setting$=[,selected$][,inactive$][,background$]
```

The menu hierarchy

The level of an item in the menu is determined by its position in the *menu hierarchy.*

```
Menu$(1)="Title"
Menu$(1,1)="Option 1"
Menu$(1,2)="Option 2"
Menu$(1,2,1)="Item 1"
```

This defines a simple menu. The structure of a menu is similar to that of an array. Each level of the menu is represented by its own dimension in the array, and is controlled using a separate version of the MENU$ command.

The first level represents the *title line* which appears at the top of your menus. It can be set using a command like:

```
menu$(n)=title$[settings$]
```

n now corresponds to the position of the title from the left of the screen, and *setting$* refers to the three optional strings which define the general appearance of the menu. It's important to define the title of your menus first as this **dimensions** the array. All other items may be created in any order you wish.

Each title is associated with a list of menu options which drop into view when the menu is selected. These form the second level of the menu structure and are defined using a second version of the MENU$ command.

```
Menu$(n,option)=Item$[setting$]
```

option holds the number of the item measured from the top left of the menu bar. There's no limit to the number of options which may be linked to a single title, other than the amount of available memory.

Each individual option can in turn be associated with its own sub menus up to a total of eight levels.

```
Menu$(n,option,sub option)=Item$[setting$]
```

Once you've created a menu it can be expanded or changed at any point in your program. Never change the current screen while you are creating a menu as this will lead to an error message.

In order to familiarise yourself with the menu hierarchy, load up the program **EXAMPLE 16.3** from the MANUAL folder.

=CHOICE *(Read a menu)*

```
item=CHOICE[(dimension)]
```

The CHOICE function checks whether an option has been highlighted on the current menu. If an item has been selected (down to the lowest level), CHOICE will return a value of -1 (true), otherwise it will be 0 (false). After you've called this function, the status of the menu will be automatically restored to 0 (false). This stops a single menu access from being accidentally detected several times.

The second form of the CHOICE command returns the option selected at the required level.

```
item=CHOICE(dimension)
```

dimension indicates the level of the menu which is to be read. As you may recall, a level number of 1 corresponds to the *title line* of the menu. Similarly the levels between 2 and 8 indicate the number of an option which has been chosen. If a menu item has not been selected, *item* will be loaded with a value of zero. For example:

```
Menu$(1)=”Menu”
Menu$(1,1)=”Option 1"
Menu$(1,2)=”Option 2"
Menu$(1,2,1)=”Option 2.1"
Menu on
```

```
Do
  If Choice Then Print Choice(1),Choice(2),Choice(3)
Loop
```

If you wanted to implement larger menus with this system, your program would need to use a long list of IF...THEN statements to deal with each and every possibility. This would cause a small but significant delay in your program while the menus were being read. It would also make it very difficult to amend your program later.

Fortunately AMOS Basic provides you with a painless method of managing even the largest menus.

•

ON MENU PROC *(Automatic menu selection)*

ON MENU PROC proc1[,proc2....]

Each title in your menu can be assigned its own procedure which will be executed automatically whenever an option is selected by the user. The action of this command is similar to the following lines of AMOS Basic code:

```
If CHOICE
   If CHOICE(1)=1
     Proc1
   Endif
   If CHOICE(1)=2
     Proc2
  Endif
:   :   :
:   :   :
Endif
```

However there is one crucial difference between the ON MENU command and the above instructions. ON MENU is performed 50 times a second using interrupts and does not affect the overall running of your program. This means that your program can be doing something totally different while the menus are being checked by the system.

Whenever the user selects a menu item the required procedure will be immediately executed with no further action on the part of your program. Your procedure can then use the CHOICE command to find which option has been chosen and perform the appropriate action.

After the procedure has concluded, your program will be returned to the instruction following the ON MENU call. Here's an example:

```
Menu$(1)="Action" : Menu$(1,1)="Count" : Menu$(1,2)="Quit"
Menu on : Rem activate menu
On Menu Proc ACTION
On Menu On : Rem Activate on menu command
Do:rem Type some characters
  X$=Inkey$ : If x$<>"" Then Print X$; : Inc W
Loop
```

```
Procedure ACTION
  Shared W
  If Choice(2)=1
     Locate 0,0 : Print "You typed ";W;" letters" : W=0
    On Menu On : Rem initialise menus
  Endif
  If CHOICE(2)=2 Then Edit
End Proc
```

There are a couple of important points to note about this example. Firstly, see how the ON MENU sequence is activated using the ON MENU ON command. This **must** be called after the menu handling procedure has finished as it's needed to restart the menuing system. Also note the use of INKEY$ rather than INPUT. The INPUT command will halt the menu checks while you are entering a line. All other commands can be used without problems, including WAIT, WAIT VBL and WAIT KEY. For a further example see **EXAMPLE 16.4**.

ON MENU GOSUB *(Automatic menu selection)*

ON MENU GOSUB label1[, label2,...]

ON MENU GOSUB enters one of a list of subroutines depending on the option which has been selected by the user. Once you've called this command and created your subroutines, the menus will be checked automatically 50 times a second.

Note that each title on the menu line is handled by its own individual subroutine. This differs from its AMIGA Basic equivalent which controls the entire menu with just a single routine.

After using this command you should activate the menuing system with a call to the ON MENU instruction. The menus must be reinitialised in this way before jumping back to the main program with RETURN. Also note that *label* **may not** be replaced by an expression as the label will only be evaluated once when the program is run. See ON MENU PROC and ON MENU ON/OFF.

ON MENU GOTO *(Automatic menu selection)*

ON MENU GOTO label 1[, label 2,...]

This command has now been superceded by the more powerful ON MENU PROC and ON MENU GOSUB instructions. It's intended to provide compatibility with programs written in STOS Basic. Whenever a menu is selected, the program will jump to the appropriate label. See ON MENU PROC, ON MENU GOSUB.

ON MENU ON/OFF *(Activate/Deactivate automatic menu selection)*

ON MENU ON

ON MENU ON activates the automatic menuing system created by the ON MENU PROC/ GOSUB/GOTO commands. After a sub-routine has been accessed in this way, the system will be **disabled**. So it's vital to reactivate the system with ON MENU ON before returning to the main program.

```
ON MENU OFF
```

This temporarily freezes the automatic menuing system. It's useful when your program is executing a procedure which needs to be performed without interruptions – such as loading and saving information to the disc. The menus can be reactivated using ON MENU ON.

ON MENU DEL *(Delete the labels used by ON MENU)*

```
ON MENU DEL
```

This erases the internal list of labels or procedures created by the ON MENU commands. You can now redirect your menus to another part of your program using a further call to ON MENU. **Warning!** Only use this command after you've deactivated the menus with ON MENU OFF.

Keyboard shortcuts

Despite the undoubted appeal of menus, some users prefer to call up the options of a program straight from the keyboard. Although menus are certainly easy for beginners, once you've Familiarised yourself with a program it can be much faster to call up an option from the keyboard.

AMOS Basic allows you to assign a keyboard shortcut to any of your menu items. These keystrokes are interpreted exactly as if the user had accessed the equivalent option from the menu. They can be used with any of the AMOS Basic menuing commands, including ON MENU.

MENU KEY *(Assign a key to a menu item)*

```
MENU KEY(,,) TO c$
MENU KEY(,,) TO scan[,shift]
```

This allows you to assign any key to any item in a previously defined menu. The only restriction is that the item you have specified **must** be at the bottom level of your menu. So you can't use a shortcut to select a sub menu as each command must correspond to a single option in the menu.

c$ is a string containing a single character which is to be assigned to the menu option. Any additional characters in the string will be ignored.

Each key on the Amiga's keyboard is assigned its own individual *scancode*. By using this code you can assign keys to a menu which have no Ascii equivalents. Here is a list of scancodes which can be used with your menus.

Scancode	Keys
80-89	Function keys F1-10
95	Help
69	Esc

shift is an optional bitmap which allows you to check for control key combinations such as Alt+Help or Control+D. The format of *shift* is:

Bit	Key Tested	Notes
0	Left Shift key	Only one shift key can be tested at a time
1	Right Shift key	
2	Caps Lock	Either ON or OFF
3	Control (Cntrl)	
4	Left Alt	
5	Right Alt	
6	Left Amiga	This is the Commodore key on some keyboards
7	Right Amiga	

Note that if you set more than a single bit in this pattern, you'll have to press several keys simultaneously to call up your menu item. Any of these short-cuts can be deactivated by using MENU KEY with no parameters. For example:

```
Menu Key(1,10):rem Turns off short cut assigned to item (1,10)
```

With the help of the MENU KEY command, adding shortcuts to a menu is a trivial operation, so you are strongly recommended to include them as standard in your programs. Here is an example that checks for the Amiga's 10 function keys:

```
Menu$(1)=" Function Keys "
For A=1 To 10
   OPT$=" F"+Str$(A)+" "
   Menu$(1,A)=OPT$
   Menu Key(1,A) To 79+A
Next A
Menu On
Do
   If Choice Then Print "You just pressed function key ";Choice(2)
Loop
```

Menu control commands

MENU ON *(Activate a menu)*

MENU ON [bank]

MENU ON activates a menu which has been previously defined in your program. The menu will be displayed when the user next presses the right mouse button, and the options can be selected in the usual way. If a *bank* number is included with the instruction, then the menu will be taken from the appropriate memory bank. See MAKE MENU BANK for more details.

MENU OFF *(Temporarily deactivate a menu)*

MENU OFF

This is the exact opposite of the MENU ON command. It temporarily freezes the action of the entire menu. The menu can be restarted at any time using MENU ON.

MENU DEL *(Delete one or more menu items)*

MENU DEL erases the selected menu from the Amiga's memory and restores the space to the rest of your program. There are two possible forms of this command.

MENU DEL

Erases the **entire** menu. **Warning!** This command is irrevocable!

MENU DEL(,,)

Deletes just a section of the menu. The (,,) parameters contain a list of up to eight values separated by commas. These indicate the precise position of the item in the menu hierarchy. For example:

```
Menu Del(1) : Rem Delete title number 1
Menu Del(1,2) : Rem Erase option 2 of title 1
```

MENU TO BANK *(Save the menu definitions in a memory bank)*

MENU TO BANK n

This instruction allows you to save an entire menu tree into memory bank *n*. If bank *n* already exists,you'll get a *bank already reserved error.*

Once you've stored a menu in this way, it will be saved automatically along with your Basic program. By storing your menu definitions in a memory bank, you can reduce the size of your program listings significantly. This will free valuable space in the editors memory, and will allow you to write longer Basic programs using exactly the same amount of memory.

BANK TO MENU *(Restores a menu definition saved in a menu bank)*

BANK TO MENU n

Sets up a menu definition from menu data stored in bank number *n.* Your menu will be restored to exactly the same state as it was originally saved. If the menu is complex, this process may take a little time. To activate your new menu call the MENU ON instruction.

MENU CALC *(Recalculate a menu)*

MENU CALC

One of the nicest features of AMOS menus is that they can be easily changed during the course of a program. After you've created your initial definition you can add new items and replace existing options at will.

All your menu items are automatically repositioned when the menu is selected with the right mouse button. If your menus are extremely large this may take a little time. MENU CALC allows you to perform this process at the most appropriate point in your program, and avoid unnecessary and unwanted delays.

Note that in order to stop the user calling the menu while it's being changed, you are strongly advised to freeze the menus with MENU OFF at the start of your procedure. The menu can then be safely restarted using the MENU ON command after you've finished.

Evolving menus are particularly useful for adventure games as each location can have its own individual menu options which can be updated depending on the player's actions.

Embedded menu commands

Any menu string can optionally include a powerful set of embedded commands which allow you to customize the appearance of your menus to an incredible degree. The list of commands is enclosed between sets of round brackets () and individual instructions are separated using colons ":". For example:

```
Menu$(1)="(Locate 10,10 : Ink 1,1) Hello"
```

Each instruction consists of just two characters which can be in either upper or lower case. Anything else will be ignored completely. Most commands also require you to input one or more numbers. These numbers **must never** make use of expressions as these are not evaluated. The commands are listed below.

Note: In the *syntax* the two important characters which make up the command are in upper case and highlighted bold.

BOB *(Draw a bob)*

BOb n

The BOB command draws bob number *n* at the current cursor position. No account is taken of the hot spot of the bob. All coordinates are measured relative to the top left corner. Also note that colour zero is usually treated as transparent. This may be changed using the NOMASK command from AMOS Basic. For example:

```
Load "AMOS_DATA:Sprites/Octopus.abk"
Menu$(1)="(Bob 1) 1":Menu$(1,1)="(Bob 2) 2":Menu$(1,2)="(Bob 3) 3"
Menu on : Wait Key
```

ICON *(Draw an icon)*

ICon n

ICON draws icon number *n* at the current cursor position. Note that unlike bobs, colour zero is **not** normally transparent. See the Basic MAKE ICON MASK command for more details.

LOCATE *(Move the graphics cursor)*

LOcate x,y

The LOCATE command moves the graphics cursor to coordinates *x,y* measured **relative** to the top left corner of the menu line. Note that after an instruction the graphics cursor will always be positioned at the bottom right of the object which has just been drawn. These coordinates will also be used to determine the location of any further items in your menu like so:

```
Menu$(1)="Example ":Menu$(1,1)="Locate (Lo 50,50) in action "
Menu$(1,2)="Guess my coords"
Menu on:wait key
```

INK *(Set Ink and Paper colours)*

INk n,value

The INK command assigns the colour indexes to be used for the PEN, PAPER and OUTLINE colours. Here is a list of the various possibilities:

n	Effect
1	Set text PEN colour
2	Set PAPER colour
3	Set OUTLINE colour

SFONT *(Set font)*

SFont n

SFont sets the current font to **graphics** font number *n*. This will be used in all future menu items. Note that you **must** call GET FONTS before this instruction is executed, otherwise it can only use the two rom fonts. See **EXAMPLE 16.5**.

SSTYLE *(Set font style)*

SStyle n

This command sets the style of the current font to *n* which is a bit-pattern in the following format:

Bit	Effect
0	Set this bit to one to underline your characters

1	Selects **bold** characters
2	Activates *italic* mode

LINE *(Draw a line)*

LIne x,y

The LINE command draws a line from the current cursor position to the graphics coordinates *x,y*. See **EXAMPLE 16.6**.

SLINE *(Set line pattern)*

SLine p

SLINE sets the line style used in all subsequent LINE commands to the bit pattern held in *p*. Since there is no expression evaluation, this pattern should always be converted into decimal notation before use. A simple demonstration of the possible line styles can be found in **EXAMPLE 16.7**.

BAR *(Draw a bar)*

BAr x,y

This draws a rectangular bar from the current cursor coordinates to *x,y*. For a demonstration see **EXAMPLE 16.8**.

PATTERN *(Draw a pattern)*

PAttern n

Changes the fill pattern used by the Bar command to style *n*. For a demonstration, load **EXAMPLE 11.8** from the AMOS MANUAL folder.

OUTLINE *(Enclose bar with an outline)*

OUtline flag

OUTLINE draws a border in the current outline colour (ink 3) around all subsequent bars. A value of one activates the border and a zero removes it.

ELLIPSE *(Draw an ellipse)*

ELlipse r1,r2

ELLIPSE draws an oval with radii *r1* and *r2* at the current cursor coordinates. To draw a circle, simply set r1 equal to r2. See **EXAMPLE 16.9**.

PROC *(Call a procedure)*

PRoc NAME

The PROC instruction allows you to call any AMOS Basic procedure directly within a menu line. The called procedure must **not** include parameters, otherwise a syntax error will be generated.

This command allows you to customize the menu precisely to your own needs without having to limit yourself to the available menu commands. In order to exploit these features, you'll need to understand a little bit of theory.

At the start of your procedure the following values are held in the 68000's processor registers.

DREG(0) **X-Coord**

This holds the graphical X coordinate of the top left corner of the current menu item. Don't draw your graphics over the part of the screen to the left of this point as this will confuse the menu redrawing process and may lead to unwanted effects.

DREG(1) **Y-Coord**

Register D1 contains the Y coordinate of your menu item. As with the X coordinate you should always limit your drawing operations to the region below this point to avoid possible errors.

DREG(2) **Status of drawing operations**

This register holds the current status of the menu operations. If it contains a value of 0 (false) the menu item is being drawn. In this case you will need to load DREG(0) and DREG(1) with the coordinates of the bottom right corner of your menu zone and return from the procedure immediately.

If DREG(0) is -1 (true) you are free to perform your graphics operations used by your procedure. After you have finished you should return the coordinates of the bottom right corner of your item in DREG(0) and DREG(1) as before.

DREG(3) **Status of menu item**

D3 is loaded with a value of -1 if the menu is highlighted and the first menu string is displayed, otherwise it will contain a value of 0.

DREG(4)

D4 is set to TRUE when the menu branch is initially opened.

AREG(1) **Address of reserved zone**

This is the address of the zone created with RESERVE. It's used to allow several procedures to communicate with each other. See RESERVE for more details.

The general structure of a menu procedure is:

```
Procedure ITEM
     If DREG(2)
          X=DREG():Y=DREG(1)
          ...Draw the item...
     Endif
     DREG(0)=BX:Rem X Coord of bottom right corner of menu item
     DREG(1)=BY:Rem X Coord of bottom corner of item
Endproc
```

The dimensions of the menu item as displayed on the screen are set using the coordinates BX and BY. These **must** be loaded into registers D0 and D1 before leaving your procedure as they are needed to create the final menu bar.

While inside your procedure you can perform most AMOS instructions including other procedures. But some instructions are absolutely forbidden! If you use these commands you won't get an error message but your AMIGA may crash unexpectedly.

- **Never** change the current screen inside a menu. This will almost certainly crash your Amiga completely!
- Don't set or reset a screen zone.
- Avoid using instructions such as WAIT, WAIT KEY or INPUT, or INKEY$ which halt the action of your program.
- Disc operations are absolutely forbidden!
- Any error trapping in your procedure will be ignored. If an error occurs, the menu will be closed and your program will return to the editor.

Used with caution, the PROC command can produce some mind-blowing effects. For a demonstration, load **EXAMPLE 16.10** from the AMOS MANUAL folder.See MENU CALLED.

RESERVE *(Reserve a local data area for a procedure)*

REserve n

Reserves *n* bytes of memory for this menu item. This area can be accessed from within your menu procedure using the address held in AREG(1). The data area you have created is common to all the strings in the current menu object. It can be used to exchange parameters between the various procedures called by a menu item.

MENU CALLED *(Redraw a menu item continually)*

MENU CALLED(,,)

The MENU CALLED command automatically redraws the selected menu item 50 times a second whenever it is is displayed on the screen. It's usually used in conjunction with a menu procedure to generate animated menu items which change in front of your eyes.

In order to make use of this function, you first need to define a menu procedure, using the principles outlined above. Then add a call to this procedure in the required title strings using an embedded PRoc command. Finally activate the updating process with a call to MENU CALL. When the user displays the chosen item, your procedure will be repeatedly accessed by the menuing system.

Since your procedure will be called fifty times a second, it should obviously return back to the menu as quickly as possible. This will allow enough time for the rest of the menu to be successfully updated.

Also note that your embedded procedure can safely animate your item using either bobs or sprites. However, as the menu items are **not** double buffered, your bobs may flicker slightly on the screen. So it may be better to use computed sprites for this purpose instead. Another approach is to draw you display with the standard AMOS graphics commands. An example of this can be seen in **EXAMPLE 16.11** in the MANUAL folder.

MENU ONCE *(Turns off automatic redrawing)*

MENU ONCE(,,)

MENU ONCE turns off the automatically updating system started using the MENU CALLED command. From now on, each menu item will only be redrawn once when the menu is called on the screen.

```
Menu Once(1,1)
```

Alternative menu styles

Normally the titles of a menu are displayed as a horizontal line and the options are arranged below it in a vertical menu bar. If you want to create something a little unusual, you can change the format of each level of your menu using the following three instructions:

MENU LINE *(Display a menu as a horizontal line of items)*

MENU LINE level
MENU LINE(,,)

The MENU LINE command displays the menu options at the requested level in the form of a horizontal line. This menu line starts from the left-hand corner of the first title and stretches to the bottom right corner of the last.

MENU LINE level

Defines the menu style of an entire level of your menu. This should only be called during your menu definitions.

MENU LINE (,,)

Normally one would only use the *level* version for this command. Setting individual items to Line and Bar can give bizarre results, but this may be useful for something!

MENU TLINE *(Display a menu as a total line)*

MENU TLINE level
MENU TLINE(,,)

MENU TLINE displays a section of the menu as a *total line* stretching from the very left of the screen to the very right. The entire line will be drawn even when the first item is in the middle of the screen.

level is a number ranging from 1 to 8 which specifies the part of the menu to be affected. This is the standard form of the instruction, and should be called during your menu definitions as otherwise it will have no effect.

You can also change the appearance of a menu after its been created using a second form of this command. For example:

```
Menu Line(1,1) : Rem Displays menu 1,1 as a line
```

MENU BAR *(Display a section of the menu as a bar)*

MENU BAR level
MENU BAR(,,)

This displays the selected menu items in the form of a vertical bar. The width of this bar is automatically set to the dimensions of the largest item in your menu.

level is a number which indicates which part of the current menu definition is to be affected. As a default this option is used for *levels 2 to 8* of your menu. Note that this form of the MENU BAR instruction may only be used during your programs initialisation phase. If you call it after a menu has been activated, it will have absolutely no effect.

(,,) is a list of parameters which allow you to change the style of your menus once they've been installed. Here's an example of Menu Bar and Menu Tline:

```
FLAG=0
SET_MEN
Do
  If Choice and Choice(1)=2 and Choice(2)=1 Then ALTER
Loop
Procedure SET_MEN
  Menu$(1)=" Bar demo " : Menu$(2)=" Select Below "
  Menu$(1,1)=" I do nothing! "
  Menu$(2,1)=" Yes, press on me! "
  Menu On
End Proc
Procedure ALTER
  Shared FLAG
  Menu Del
  If FLAG=0 Then Menu Bar 1 : FLAG=1 Else Menu Tline 1 : FLAG=0
  SET_MEN
```

```
End Proc
```

MENU INACTIVE *(Turn off menu item)*

MENU INACTIVE level
MENU INACTIVE(,,)

As its name suggests, MENU INACTIVE deactivates a series of options on your menu. Any subsequent attempts to select these items will be completely ignored. *level* allows you to deactivates an entire section of the menu and you can also deactivate individual menu options with the parameters in (,,). These indicate the precise position of your item in the current menu hierarchy.

Note that the menu items you've turn off with the instruction will be immediately replaced by the INACTIVE$ string you specified during your original menu definition. If this was omitted, any unavailable menu options will be shown in *italics.*

MENU ACTIVE *(Activate a menu item)*

MENU ACTIVE level
MENU ACTIVE(,,)

MENU ACTIVE simply reverses the effect of a previously MENU INACTIVE command. *level* chooses an entire menu level to be restarted. (,,) activates a single item on your menu.

After you've called this instruction, the selected options will be automatically redisplayed using their original title strings.

Moveable menus

AMOS menus can be displayed at any point on the Amiga's screen. Menus can be moved either explicitly by your program or directly by the user.

MENU MOVABLE *(Activate automatic menu movement)*

MENU MOVABLE level
MENU MOVABLE(,,)

MENU MOVABLE informs the menuing system that the menu items at *level* may be moved directly by the user – this is the default.

The second form of this command allows you to set the status of each individual item in the menu. The parameters between the brackets can indicate any position in the menu hierarchy.

Any menu may be repositioned by moving the mouse pointer over the **first** item in the menu and pressing the left mouse button. A rectangular box will now appear around the selected menu item, and this may be moved to any point on the current screen. When you release the left button the menu will be redrawn at the new position along with all the associated menu items.

Note that this command does not allow you to change the arrangement of any items below this level. If you want to manipulate the individual menu options you'll need to use a separate MENU ITEM command. See **EXAMPLE 16.12** for a demonstration of this system.

MENU STATIC *(Fix a menu into place)*

```
MENU STATIC level
MENU STATIC(,,)
```

MENU STATIC defines the menu at *level* to be immovable by the user.

One problem with moveable menus is that the memory they consume will change during the course of a program. If your menus are particularly large, or if memory is running tight, this can cause real problems as a single careless action by the user will abort your program with an *out of memory* error. With the help of the MENU STATIC command you can avoid this difficulty completely.

MENU ITEM MOVABLE *(Move individual menu options)*

```
MENU ITEM MOVABLE level
MENU ITEM MOVABLE(,,)
```

This command is similar to MENU MOVABLE except that it allows you to re-arrange the various options in a particular level. So all the items in a menu bar may been individually repositioned by the user.

Normally it is illegal to move the items outside the current menu bar, but this can be overridden using the MENU SEPARATE command.

In order for the menu items to be movable, the **whole** menu bar must also be movable. So if you fix the MENU into place with MENU STATIC, this command will have no effect. Additionally you can't move the first item in the menu bar as this will move the entire line. Another side effect is that moving the **last** menu item will permanently reduce the size of your menu bar. There are two possible solutions:

- Enclose your entire bar with a rectangular box like so:

```
Menu$(1,1)=,,,”(Bar 40,100)(Loc 0,0)”
```

Where MENU$(1,1) is the first item in your current bar.

- Set the last item into place with MENU ITEM STATIC.

MENU ITEM STATIC *(Static menu item)*

```
MENU ITEM STATIC level
MENU ITEM STATIC(,,)
```

This command locks one or more menu items firmly into place and is the default setting.

MENU SEPARATE *(Separate a list of menu items)*

MENU SEPARATE level
MENU SEPARATE(,,)

MENU SEPARATE tells AMOS to separate all the items in the current level. Each item in your menu is treated completely independently from the previous one. If you haven't defined a background string, each item will be offset by two pixels from the one above. This creates an attractive stepped effect which can be removed by editing the menu with the MENU Accessory.

The optional parameters to this instruction allow you to split a menu bar at any point in the line. Once you've separated an item it will be affected by the MENU MOVABLE commands rather than the ITEM instructions.

MENU LINKED *(Link up a set of menus)*

MENU LINKED level
MENU LINKED(,,)

This links one or more menu items together. It's the opposite of the MENU SEPARATE INSTRUCTION.

=MENU X *(Return the graphical X coordinate of a menu item)*

x=MENU X(,,)

The MENU X function allows you to retrieve the position of a menu item relative to the previous option on the screen. You can use this information to implement powerful menus such as the one found in **EXAMPLE 16.13**

=MENU Y *(Return the graphical Y coordinate of a menu item)*

x=MENU Y(,,)

Returns the Y coordinate of a menu option. Note that all coordinates are measured relative to the previous item. So this is **not** a standard screen coordinate.

Moving a menu within a program

MENU BASE *(Move the starting point of a menu)*

MENU BASE x,y

This command moves the starting point of the first level of your menus to the absolute screen coordinates *x,y*. All subordinate menu items will be displayed at their current positions relative to the top of your menu. See **EXAMPLE 16.14** for a demonstration of the MENU BASE command in action.

SET MENU *(Move a menu)*

SET MENU (,,) TO x,y

SET MENU sets the coordinates of the top left corner of a menu item. These coordinates are measured **relative** to the previous level. The starting point for the entire menu (coordinates 0,0) may be set with the MENU BASE command.

All the levels of the menu below your menu will also be moved by this instruction. Their relative positions will be unchanged. Since *x,y* can be negative it's possible to arrange the items in a menu bar in the form of a control panel – see **EXAMPLE 16.15**.

Displaying a menu at the cursor position

MENU MOUSE *(Display the menu under the mouse)*

MENU MOUSE ON/OFF

The MENU MOUSE features automatically display all menus starting from the current position of the mouse cursor. The mouse coordinates are added to the MENU BASE to get the final position, so it's possible to place the menu a fixed distance away from the mouse pointer if required. See **EXAMPLE 16.16**.

17: Sound and music

The Amiga's sound system is capable of generating stereo sound effects which would have been unheard of just a few years ago. The results are impressive even through your TV speaker, but when you connect your Amiga to a Hi-Fi, the sounds can actually shake your room!

As you would expect from AMOS, we've come a long way since the humble BEEP command. In fact, we've provided everything you need to incorporate mind-blowing sound effects in your own games. All the AMOS sound commands are performed independently of your Basic programs. So your sound tracks can be played continuously, without affecting the quality of the game-play in the slightest.

Samples may be created using any of the available sampling cartridges and can be replayed with a simple SAMPLAY instruction. Each sample can be output in a variety of speeds, and may be looped repeatedly. It's even possible to play a sample as a musical note.

Music can be converted over from a separate package such as SONIX, SOUNDTRACKER or GMC. The AMOS Music system is intelligent and will automatically stop when a sound is played through the current channel, thus allowing you to effortlessly combine both samples and music in the same sound channel, without the risk of unwanted interference effects.

Each song can incorporate up to 256 separate instruments; the only limit to the number of songs is the amount of available memory. In order to keep the memory overhead down to an absolute minimum, all tunes are built up out of a number of separate patterns. Once a pattern has been created, it can be accessed anywhere in your music using just a couple of bytes. By defining just a few key patterns, you can therefore create dozens of different tunes without running short of memory.

The best thing about the AMOS music system however, is that it's expandable. The entire source code is supplied on the data disc for you to examine or change. So you won't be left out in the cold by any future developments on the Amiga's music scene.

Simple sound effects

We'll start off with a run down of the built-in sound effects included in AMOS Basic. These are the AMOS equivalent to the Amiga Basic BEEP command.

BOOM *(Generate a noise sounding like an explosion)*

BOOM

Kapow! You're dead! Use BOOM to add the appropriate stereo sound effect in your games.

Traditionally this type of "White Noise" has been extremely difficult on the Amiga, but AMOS uses a clever interrupt system to create a realistic explosion effect. Example:

```
Boom : Print "You're DEAD!"
```

SHOOT *(Create a noise like a gun firing)*

SHOOT

The SHOOT command generates a simple gunshot effect. Like BOOM, SHOOT does not halt your program in any way. So if you're firing several successive shots, you may wish to add a small delay using WAIT.

```
Shoot : Wait 6 : Shoot : Print "You're DEAD!"
```

BELL *(Simple bell sound)*

BELL [f]

BELL produces a pure tone with frequency *f*. *f* sets the pitch of the note, from 1 (very deep) to 96 (very high). Example:

```
Bell : Wait 40 : Rem Wait for ring to complete
For F=1 To 96
  Bell F:Wait F/10+1 : Rem Vary the delay along with the frequency
Next F
```

Sound channels

The Amiga's hardware can effortlessly play up to four different sounds simultaneously. This allows you to add attractive harmonics to your sound effects.

Each sound can be output through one of four *voices* numbered from 0 to 3. You can think of these voices as separate instruments which can independently play their own sequence of notes, samples, or music. All four voices are internally combined to generate the final sound you hear through your speaker system.

The AMOS sound instructions will happily play your sounds using any arrangement of voices you like. All AMOS sound commands use a standard way of entering your voice settings. Each voice is assigned a particular bit in a VOICE parameter like so:

Bit 0-> Voice 0
Bit 1-> Voice 1
Bit 2-> Voice 2
Bit 3-> Voice 3

To activate the required voices, set the appropriate bits to 1. Here's a list of common values to make things a little easier.

Value	**Voices used**	**Effect**
15	0,1,2,3	Uses all four voices.
9	0,3	These voices are combined together and output to the left speaker.
8	3	
6	2,4	Played through the RIGHT speaker.
4	2	
2	1	
1	0	

In order to do justice to the resulting sound effects, you'll almost certainly need to connect your Amiga to a hi-fi system of some sort. Most TVs are just not capable of reproducing the full range of sounds which can be generated by the Amiga's amazing hardware.

VOLUME *(Change the sound volume)*

VOLUME [v,] intensity

VOLUME changes the volume of the sounds which are to be played through one or more sound channels.

Intensity refers to the loudness of this sound. It can normally range from 0 (silent) to 63 (very loud). As a default, the volume is set to the same intensity for all four of the available voices. The new volume will be used for all future sound effects, including music.

The *v* parameter lets you change the volume of each voice independently. *v* now indicates which combination of voices are to be regulated. This second option is only used by the sound effects. It has no affect on any music you are playing. The voices are selected using a bit map in the standard format, with each bit representing the state of a single sound channel. If the bit is set to 1, then the volume of this voice will be changed, otherwise it will be completely unaffected. Examples:

```
Volume %0001,63 : Boom : Wait 100 : Rem Set Channel 1 to a volume of 63
Volume %1110,10 : Boom : Wait 50 : Rem Channels 2,3,4 have a volume of 10
Play 40,0 : Wait 30
Volume 50 : Play 40,0
```

Sampled sound

If you had to generate all the sound effects you need, directly inside your computer, you would be faced with an impossible task. In practice, it's often much easier to take a real sound from an external source, such as a tape recorder, and convert it into a list of numbers which can be held in your computer's memory.

Each number represents the volume of a particular sample of the sound. By rapidly playing these values back through the Amiga's sound chips, you can recreate a realistic impression of the original sound. This technique forms the basis of the sampled sound effects found in most modern computer games.

If you want to create your own samples, you'll be forced to buy a separate piece of hardware known as a SAMPLER CARTRIDGE. Although these cartridges are great fun, they're certainly not essential. AMOS Basic is perfectly capable of playing any existing sound sample, without the need for any expensive add-ons.

Currently there are hundreds of sound effects available from the public domain, covering the vast majority of the effects you'll need in your games. We've even included a selection of useful samples on the AMOS data disc for you to experiment with.

SAM PLAY *(Play a sound sample from the AMOS sample bank)*

SAM PLAY s
SAM PLAY v,s
SAM PLAY v,s,f

The SAM PLAY instruction plays a sampled sound straight through your loudspeaker system. All samples are normally stored in memory bank number five, but this may be freely changed using the SAM BANK command.

s is the number of the sample which is to be played. There's no limit to the number of samples you can store in a bank other than the available memory. If you want to use your own samples with this instruction, you'll need to incorporate them into an AMOS memory bank. Full details can be found towards the end of this section.

v is a bit-map containing a list of voices your sample will use. As usual, there's one bit for each possible voice. To play your samples through the required voice, simply set the relevant bit to 1. See the previous explanation of SOUND CHANNELS for more information.

f holds the playback speed of your sample, measured in *hertz*. This specifies the number of samples which are to be played each second. Typical sample speeds range from 4000, for noises such as explosions, to 10000 for recognisable speech effects. By changing the playback rate, you can freely adjust the pitch of your sound over a large range. So a single sample can be used to generate dozens of different sounds.Examples:

```
Rem Load the sample bank with some samples from the AMOS DATA disc
Load "AMOS_DATA:SAMPLES/~~SAMPLE_DEMO.ABK~~"   SAMPLES.ABK
For S=1 to 11
  Locate 0,0 : ? "Playing sample ";S
  Sam Play S
Locate 0,24 : Centre"<Hit a key to continue>" : Wait Key : Cline
Next S
Wait Key
Sam Play 1,11 : Wait 5 : Sam Play 2,11 : Rem simple echo effect
Wait key
Sam Play 1,1,2000 : Rem Low Pitch
Wait Key
Sam Play 1,1,15000 : Rem High pitch
```

A further demonstration of this command can be found in **EXAMPLE 17.1** in the MANUAL folder.

SAM BANK *(Change the current sample bank)*

SAM BANK n

SAM BANK assigns a new memory bank to be used for your samples. All future SAM PLAY instructions will now take their sounds directly from this bank.

It's possible to exploit this feature to hold several complete sets of samples alongside each other. You can then flick between these samples at any time, with just a simple call to the SAM BANK command.

SAM RAW *(Play a sample from memory)*

SAM RAW voice,address,length,frequency

SAM RAW plays a raw sample stored anywhere in the Amiga's memory. *voice* is a bit-pattern in standard format which specifies the list of voices your sample is to use. Each bit in the pattern selects a single channel to be played (see sound channels).

address holds the address of your sample. Normally, this will refer to the inside of an existing AMOS memory bank. *length* contains the length of the sample you wish to play. *frequency* indicates the sample speed to be used for the playback (in samples per second or Hz). This may be very different to the rate at which the sample was originally recorded.

SAM RAW lets you play standard Amiga samples straight through your loudspeaker, without the need to create a special memory bank (see *Creating a sample bank*). It's now your responsibility to manage your samples in memory, and enter the sample parameters by hand. SAM RAW is great for browsing through files from your disc collection. Use BLOAD to hold a file in a bank and then use SAM RAW to play the data. With luck you should come across some interesting sounds. Examples:

```
Reserve as work 10,55000
Bload "Samples/Samples.abk",start(10)
Sam Raw 15,start(10),length(10),10000
```

SAM LOOP *(Repeat a sample)*

SAM LOOP ON/OFF

The SAM LOOP directive informs AMOS Basic that all subsequent samples are to be repeated continuously. Examples:

```
Rem Load the sample bank with some samples from the AMOS DATA disc
Load "AMOS_DATA:SAMPLES/~~SAMPLEDEMO~~.ABK"
Sam Loop On                        SAMPLES.Abk
For S=1 to 11
  Locate 0,0 : Print "Playing sample ";S
  Sam Play S
  Locate 0,24 : Centre"<Hit a key to continue>" : Wait Key : Cline
Next S
Sam Loop Off
```

This looping effect can be deactivated with a simple call to the SAM LOOP OFF command.

Creating a sample bank

If you're intending to play your own samples using SAM PLAY, you'll first need to load them into a memory bank. This can be achieved with the SAMMAKER program supplied on the AMOS data disc.

On start-up, SAMMAKER presents you with a standard AMOS file selector. Enter the filename of the first sample to be stored in your new bank, and press RETURN. If AMOS can't find the sampling rate, you'll be asked to enter it directly. It doesn't really matter if you make a mistake at this point, as you can safely replay your samples at any speed you like.

After a short delay, you'll be prompted for the next sample to be installed into the bank. When you've reached the end of your samples, type SAVE at the file-selector to save your samples onto the disc. You'll be automatically prompted for the destination filename of your new bank. This can now be entered into AMOS Basic using the LOAD command like so:

```
                   s
Load “sample.abk”
Load “sample.abk”,6 : Rem load samples into bank 6
                   s
```

Music

The AMOS music system allows you to easily add an attractive backing track to your games. Music can be created from a variety of sources, including GMC, SOUNDTRACKER or SONIX.

In order to convert these musics into the special AMOS format, you'll need to use one of the translation programs included on the AMOS data disc.GMC music should have been saved using the SAVE DATA icon, as this copies both the music and the instrument definitions into a single large data file.

MUSIC *(Play a piece of music)*

MUSIC n

The AMOS MUSIC command starts a piece of music from the music bank (three). This music will be played independently of your Basic program, without affecting it in the slightest.

Normally, it's possible to store several complete arrangements in the same bank. Each composition is assigned its own individual music number. The only exception to this rule is music created by GMC, which only allows you to place one song in the bank at a time. Example:

```
Rem Load a piece of music from the AMOS data disc
Load “MUSIC/musicdemo.abk”
Music 1
```

The AMOS music system is intelligent, and will automatically suspend your music for the duration of any subsequent sound effects on the current channel. When the sound has finished, your tune will be restarted from its previous position.

Up to three separate tunes can be started at a time. Each new music command stops the current song, and pushes its status onto a *stack*. Once the song has concluded, the old music will commence from where it left off.

MUSIC STOP *(Stop a single section of music)*

MUSIC STOP

MUSIC STOP halts the current piece of music. If another music is active, it will be restarted immediately.

MUSIC OFF *(Turn off all music)*

MUSIC OFF

The MUSIC OFF command deactivates your music completely. In order to restart it, you'll need to execute your original series of MUSIC instructions again from scratch.

TEMPO *(Change the speed of a sample of music)*

TEMPO s

TEMP modifies the speed of any tune which is currently being played with the MUSIC command. *s* is the new speed, and can range from 1 (very slow) to 100 (very fast). Not all instruments are capable of playing at this maximum speed, however. The practical limit is closer to 50.

For a demonstration, place the AMOS data disc into the current drive and type:

```
Load "AMOS_DATA:MUSIC/musicdemo.abk" : Rem Load music
Music 1 : Rem Play music 1
Tempo 35 : Rem Set music playing very fast
Tempo 5 : Rem Start music playing very slow
```

Note that music created with GMC often contains labels which set the tempo directly inside the arrangement. These labels will override the tempo settings within AMOS Basic. So it's not advisable to use them in your own music.

MVOLUME *(Set the volume of a piece of music)*

MVOLUME n

This changes the volume of the entire piece of music to intensity *n. n* can range from 0 (silent) to 63 (very loud).

VOICE *(Activate one or more voices of a piece of music)*

VOICE mask

VOICE activates one or more voices of the music independently. Usually each voice will contain its own separate melody which will be combined through your speakers to generate the eventual music.

mask is a bit mask in the normal AMOS format which specifies which voices you wish to play. Each bit represents the state of one voice in the music. If it's set to 1, the voice will be played, otherwise it will be totally unused. Examples:

```
Load "MUSIC/musicdemo.abk" : Rem Load music
Music 1 : Rem Play music 1
For V=0 To 15
Locate 0,0 : Print "Voice ";V
Voice V
Wait 100
Next V
Direct
```

```
Voice %0001 : Rem Activate Voice 0
Voice %0010 : Rem Voice 1
Voice %1001 : Rem voices 3 and 0
Voice %1111 : Rem Voice 4
```

=VUMETER *(Volume meter)*

s=VUMETER(v)

The VUMETER function tests voice *v* and returns the volume of the current note which is being played by your music. *s* is an intensity value between 0 and 63. *v* is the number of a single voice to be checked (from 0-3).

Using this function, you can make your sprites dance to a piece of music! Load **EXAMPLE 17.2** for a demonstration.

Note there's also an AMAL version of this instruction which allows you to create real-time VU meters using interrupts. See the section on the VU command for more information.

Playing a note

PLAY *(Play a note)*

PLAY [voice,] pitch,delay

Plays a single note through the loudspeaker of your TV or Hi-FI. *Pitch* sets the tone of this sound, ranging from 0 (low) to 96 (high). Rather than just being an arbitrary number, each pitch is associated with one of the notes (A,B,C,D,E,F,G). This can be seen from the following table.

				Octave				
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Note				**Pitch**				
C	1	13	25	37	49	61	73	85
C#	2	14	26	38	50	62	74	86
D	3	15	27	39	51	63	75	87
D#	4	16	28	40	52	64	76	88
E	5	17	29	41	53	65	77	89
F	6	18	30	42	54	66	78	90
F#	7	19	31	43	55	67	79	91
G	8	20	32	44	56	68	80	92
G#	9	21	33	45	57	69	81	93
A	10	22	34	46	58	70	82	94
A#	11	23	35	47	59	71	83	95
B	12	24	36	48	60	72	84	96

It should be apparent that the notes go up in a cycle of 12. This cycle is known as an octave.

The optional *voice* parameter allows you to play your notes through any combination of the Amiga's four voices. As usual it's a bit-map in the format:

Bit 0-> Voice 0
Bit 1-> Voice 1
Bit 2-> Voice 2
Bit 3-> Voice 3

Setting a bit to a value of 1 plays the note through the relevant voice.

Delay sets the length of the pause between the play command and the next Basic instruction. This allows you to play each note before preceding with the next one.

A delay of zero starts a note and immediately jumps to the next Basic instruction. By playing several notes one after another, you can easily generate some attractive harmonic effects. Examples:

```
Play 1,40,0 : Play 2,50,0 : Rem Play with no delay
Wait Key
Play 1,40,15 : Play 2,50,15 : Rem See the effect of delay
Rem Play a random sequence of notes
Do
  T=Rnd(96) : V=Rnd(15) : Play V,T,3
Loop
```

PLAY is not just limited to pure notes incidentally. It's also possible to assign complex waveforms to the sound generator using the powerful WAVE and NOISE commands.

Waveforms and envelopes

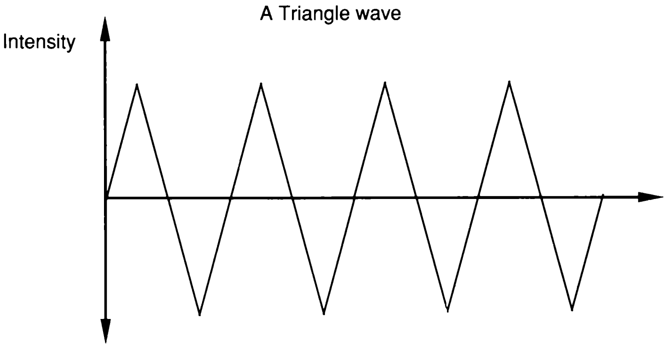
SET WAVE *(Define a waveform)*

SET WAVE wave,shape$

The SET WAVE instruction provides you with the ability to define your very own instruments for use with the AMOS Basic PLAY instruction. The sound of your instrument depends on the shape of a waveform held in the Amiga's memory.This forms a template which is repeated to produce your final note.

wave is the number of the waveform you wish to define. Allowable wave numbers start from 2 onwards. That's because waves zero and 1 are already installed. Wave zero holds a random noise pattern for producing explosion effects. Wave one is a smooth sine wave and generates the pure tones used by the standard PLAY instruction.

The shapes of your waveforms are set using a list of 256 numbers which are entered using the SHAPE$ parameter. Here's an example of one of these waveforms for you to examine.



Each number represents the intensity of an individual section of the waveform. This is equivalent to the height of just one point in the above graph.

Possible values for the intensity range from -128 to 127. Since AMOS strings are only capable of holding **positive** numbers (0-255), you'll need to convert your negative values into a special internal format before use. The required value can be calculated by simply adding 256 to the negative numbers in your list. So -50 would be entered as:

-50+256=206

Here's a program which demonstrates how the triangular wave in the previous diagram could be created in AMOS Basic.

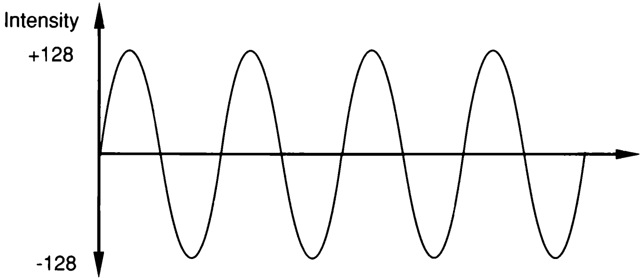
```
S$="" : Rem clear waveform string
For I=-128 To 127
  X=I : If X<0 Then Add X,256
  S$=S$+Chr$(X)
Next I
Set Wave 2,S$
```

Before playing your waveform you have to tell AMOS Basic which channels are to be assigned to your wave. This can be achieved using the WAVE command. Add the following line to the previous routine

```
Wave 2 To 15 : For S=10 to 60 : Play S,10 : Next S
```

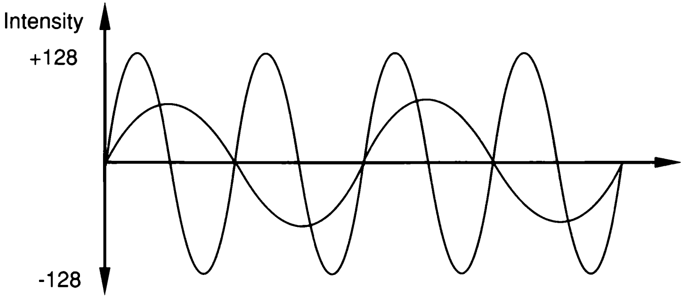
The best way to reproduce the effect of a real instrument is to combine several SINE waves together. An example of one of these sine waves can be seen in the diagram below.

A Sine wave



Adding several of these waves together, with different sizes and separate starting points, produces waves in the following pattern.

Complex waves



This generates the smooth harmonics needed for your notes. Here's an example:

```
 SHAPE$=”” : Degree
 For S=0 To 255
   V=Int((Sin(S)/2+Sin(S*2+45)/4)*128)+127
   SHAPE$=SHAPE$+Chr$(V)
 Next S
Set Wave 2,SHAPE$ : Wave 2 To 15
For N=10 To 60 : Play N,10 : Next N
```

WAVE *(Assign a wave to one or more sound channels)*

WAVE w To v

WAVE assigns wave number *w* to one or more sound channels. *v* contains a bit-map in the standard format. If a bit in the pattern is set to 1 then the appropriate voices are used by PLAY, otherwise they will be completely unaffected.

As a default, wave zero is reserved for the NOISE channel, and wave one contains a sine wave. Here are some examples:

```
Wave 0 To %0001 : Rem Assign wave 0 to voice zero
Play 1,40,0
Wave 0 To %1100 : Rem Use voices 3,2 for noise
Play 20,10
Wave 1 To %1111 : Rem Play pure tone through all four voices
Play 60,0
```

NOISE *(Assign a noise wave to a channel)*

NOISE To voices

NOISE applies a white noise effect (wave 0) to the selected voices. This forms the basis for a wide range of explosion and percussion effects. Load **EXAMPLE 17.3** from the MANUAL folder for a demonstration.

voices is a standard bit pattern. The first four bits represent the Amiga's four possible voices, starting from zero. If a bit is set to 1 then noise will be played this channel, otherwise this voice will be completely unchanged by this instruction. NOISE is equivalent to the command:

```
Wave 0 To voices
```

Examples:

```
Noise To 15
Play 60,0
Play 30,0
```

DEL WAVE *(Delete a wave)*

DEL WAVE n

Deletes a wave which has previously been defined using SET WAVE. *n* is the number of the wave, and starts at 2. It's impossible to delete the built-in NOISE and SINE waves using this instruction. After the wave has been erased, all voices will be reset to the standard SINE wave (default).

SAMPLE *(Assign a sample to a wave)*

SAMPLE n To voices

This is the most powerful version of all the wave commands. It assigns a sample stored in the sample bank to the current wave. Play will now take an instrument straight from the sample bank.

```
Load "SAMPLES/SAMPLE~~1~~5.abk"
Sample 1 To 15
For I=20 To 50
  Play i,50
Next I
```

As usual *voices* allows you to select a range of voices to be set by the instruction. It's a standard bit-map in the format:

Bit 0-> Voice 0
Bit 1-> Voice 1
Bit 2-> Voice 2
Bit 3-> Voice 3

Each voice can be selected by setting the associated bit to 1.

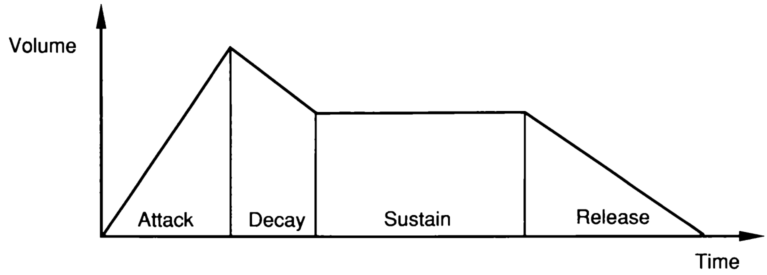
Note: The range of notes that a sample can be played with, depends on its original recording rate. If a note is too high, AMOS may not be able to play it at all. The acceptable range varies from sample to sample, but it's usually between 10 and 50.

SET ENVEL *(Create a volume envelope)*

SET ENVEL wave,phase TO duration,volume

The SET ENVEL command smoothly changes the volume of a note while it is being played. In the real world, sounds don't just spring into existence fully formed. They tend to evolve over a period of time, according to a pattern known as the volume envelope. The shape of this envelope varies depending on the type of instrument you are playing. Here's a typical example of one of these envelopes.

A typical volume envelope



This sound is split up into four phases: Attack, decay, sustain and release.

AMOS Basic allows you to define your envelopes using up to seven separate steps. Each step represents a steady change in the volume of the current note.

wave is a number of the waveform which will be affected by this instruction. It's possible to use any waveform you like for this purpose, including the built-in NOISE and SINE generators.

phase holds the number of the particular phase which is to be defined, ranging from 0 to 6.

duration specifies the length of the current step in units of a 50th of a second. This determines the apparent speed of the volume change to be generated in this phase.

volume specifies the volume which is to be reached by the end of this phase. Allowable volume levels range from 0-63.

It's important to understand that this volume is relative to the intensity you've previously set with the VOLUME command. So even if the note is quiet, the shape of the envelope will be perfectly preserved. Now for some examples:

```
Set Envel 1,0 To 200,63 : Rem Sets the length of the first step
Play 40,0 : Rem this note will be played for four whole seconds
```

As you can hear, the volume of your sound starts from zero, and increases to a maximum intensity during the length of the note. Now let's try defining something a little more complicated:

```
Set Envel 1,0 To 15,60 : Rem Slow attack
Play 40,0 : Wait Key
Set Envel 1,1 To 1,50 : Rem Brief decay
Play 40,0 : Wait Key
Set Envel 1,2 To 10,50 : Rem short sustain
Play 40,0 : Wait Key
Set Envel 1,3 To 50,0 : Rem Long release
Play 40,0
```

Finally, here's an example of a NOISE envelope:

```
Noise To 15
Set Envel 0,0 To 1000,30
Play 40,0
Wait key
Music Off : Rem Kills sound
```

Don't confuse waves and envelopes. A wave sets the frequency components of your notes, whereas an envelope simply changes their volume according to a set pattern.

Speech

Your Amiga is supplied with a powerful speech synthesiser program which can be found on the standard Workbench disc. With the help of this routine, your AMOS programs can be made to speak. Speech is especially useful in education, as many young people will respond far better to the spoken word than to boring text.

One word of caution though. Since the narrator package is independent of AMOS

Basic, we can't attest to its absolute reliability. You're unlikely to encounter any serious problems, but it's well worth treating it with a little care.

SAY *(Speak a phrase)*

SAY t$[,mode]

The SAY command is incredibly easy to use. Enter your text in normal English, concluding your phrase with a punctuation mark such as a full stop. SAY will now translate your words into an internal format and speak them directly through your loudspeaker. Example:

```
Say "AMOS Basic can really speak."
```

The first time you use this instruction, the NARRATOR device will automatically be loaded from disc. So it's vital to ensure that an appropriate disc is placed in the current drive before using this system, as otherwise you may get a Intuition style requester box.

mode toggles between two separate speech modes. As a default, your program will wait for the duration of the speech, and any music or sound effects will be temporarily suspended. Setting *mode* to a value of one activates a multitasking system which allows you to output your speech whilst AMOS is executing your program. Inevitably, this will slow down your basic routines considerably. To return your speech back to normal, set *mode* to zero. Example:

```
Do
  Input "Enter a phrase?";T$
  T$=T$+"."
  Say T$,1
Loop
```

If the narrator system cannot understand what you are attempting to speak you won't get an error message, but the command will be automatically aborted.

Also note that the narrator can occasionally get slightly confused with very short sentences. Sometimes the remainder of the previous phrase is tagged to the end of the current voice. The problem can be solved by simply adding a list of spaces to the end of your text. These will wipe out the unwanted speech data.

SET TALK *(Set speech effects)*

SET TALK sex,mode,pitch,rate

SET TALK allows you to change the type of voice which will be used by the SAY command.

sex chooses between a *male* (0) or *female* (1) voice. In all honesty, it's not a particularly realistic rendition. Better effects can be created by simply increasing the frequency of the voice using the *pitch* parameter.

mode adds a strange rhythmic pattern to the voice. This can be activated by setting *mode* to a value of 1. It's quite a pleasant effect, but hardly human. You can reset the voice to normal with a parameter of zero.

pitch changes the frequency of the voice, from 65 (low) to 320 (incredibly high).

rate specifies the speed, measured in the words per minute. Allowable word rates range from 40 to 400.

Any of the above parameters can be omitted if required. Providing you keep the commas in their normal positions, you can change any set of options independently. Here are some examples:

```
Set Talk 1,,, : Say "This is a female voice."
Set Talk ,1,, : Say "This is a female voice with intonation."
Set Talk 0,0,320, : Say "A soprano voice."
Set Talk ,,65, : Say "A deep voice."
Set Talk 0,0,100,400 : Say "Say this quickly."
Set Talk ,,,40 : Say "This is slow."
```

Filter effects

LED *(Activate a high pass filter/change power led)*

LED ON/OFF

The LED command has two completely separate actions. Not only does it toggle the POWER led on your Amiga's console, but it also controls a special *high pass* filer.

The filter changes the way high frequency sounds are treated by the system. Normally, these sounds are filtered out so as to avoid the risk of unwanted distortion effects. Unfortunately, this robs many percussion instruments of their timbre. By turning off the filter, you can recapture the essential quality of many instruments. Example:

```
Load "AMOS_DATA:MUSIC/MUSICDEMO.ABK" : Rem from the AMOS data disc.
Music 1
Do
  If mouse key=1 Then Led on
  If mouse key=2 Then Led off
Loop
```

If you experiment with this function, you'll find that some parts of the music sound much better without the filter, whereas others distort horribly when the filter is deactivated. The power led now lets you see the current filter setting at a glance. Warbling this in time to the music, can considerably improve the overall effect.

18: The keyboard

Sometime in the far distant future, we'll probably be entering our programs directly using some sort of mind link. Until then, the fastest way of entering information into a computer will be to type it in from the keyboard.

AMOS Basic provides you with dozens of useful keyboard commands. These can be used in anything from an Arcade game to an Adventure. It's even possible to write a fullly fledged wordprocessor entirely in AMOS Basic!

=INKEY$ *(Function to get a keypress)*

k$=INKEY$

The INKEY$ function checks whether the user has pressed a key, and returns its value in the string *k$*.

Note the INKEY$ command doesn't wait for your input in any way. If the user hasn't entered a character, INKEY$ will simply return an empty string "". Example:

```
Do
  Rem Try hitting some of the cursor keys
  X$=Inkey$: If X$<>"" Then Print X$;
Loop
```

INKEY$ is only capable of reading keys which return a specific Ascii character from the keyboard. Ascii is a standard code used to represent all the characters which can be printed on the screen.

It's important to realise that some keys, like the HELP button or the function keys, use a rather different format. If INKEY$ detects such a key, it will return a character with a value of zero (CHR$(0)). You can now find the the internal *scan code* of this key using a separate SCAN CODE function.

=SCANCODE *(Input the scancode of the last key input with INKEY$)*

s=SCANCODE

SCANCODE returns the internal scancode of a key which was previously entered using the INKEY$ function. This allows you to check for keys which do not produce a character from the keyboard, such as HELP or TAB. Try typing in the following small example:

```
Do
  While K$=""
    K$=Inkey$
  Wend
  If Asc(K$)=0 Then Print "You Pressed A Key With No ASCII Code."
  Print "The Scancode Is";Scancode
  K$=""
Loop
```

=KEY STATE *(Test whether an individual key has been pressed)*

t=KEY STATE(s)

Checks if a specific button has been pressed on the Amiga's keyboard. *s* is the internal scancode of the key you want to check. If this key is currently being depressed then KEY STATE will return a value of TRUE (-1), otherwise the result will be FALSE(0). Example:

```
Do
   If Key State(69)=True Then Print "Escaped!" : Rem Esc key pressed
   If Key State(95)=True Then Print "HELP!" : Rem Help Key pressed
Loop
```

=KEY SHIFT *(Return the status of the shift keys)*

keys=KEY SHIFT

KEY SHIFT returns the current status of the various control keys. These keys such as SHIFT or Alt cannot be detected using the standard INKEY$ or SCANCODE system. But you can easily test for any combination of control keys with just a single call to the KEY SHIFT function. *keys* is a bit map in the following format:

Bit	Key Tested	Notes
0	Left SHIFT key	
1	Right SHIFT key	
2	Caps Lock	Either ON or OFF
3	Control (Cntrl)	
4	Left Alt	
5	Right Alt	
6	Left Amiga	This is the Commodore key on some keyboards
7	Right Amiga	

If a bit is set to a one, then the associated button has been held down by the user. Example:

```
Centre "<Press some control keys>"
Curs Off
Do
   Locate 14,4 : Print Bin$(Key Shift,8)
Loop
```

=INPUT$(n) *(Function to input n characters into a string)*

x$=INPUT$(n)

INPUT$ enters *n* characters straight from the keyboard, waiting for each one in turn. As with INKEY$, these characters are not echoed onto the screen.

x$ is a string variable which will be loaded with your new characters. *n* holds the number of characters to be entered. Example:

```
Clear Key : Print "Type In Ten Characters"
C$=Input$(10) : Print "You entered ";C$
```

This instruction is **not** that same as the standard INPUT command. The two instructions are completely different. Also note that there's a special version of INPUT$ which can be used to read your characters from the disc.

WAIT KEY *(Wait for a keypress)*

WAIT KEY

Waits for a single keypress. Example:

```
Print "Press A Key" : Wait Key : Print "Key Pressed"
```

KEY SPEED *(Change key repeat speed)*

KEY SPEED lag,speed

KEY SPEED lets you tailor the speed of the keyboard to your own particular taste. The new speed will be used for every part of the AMOS system, including the editor.

lag is the time in 50ths of a second between pressing a key, and the start of the repeat sequence.

speed is the delay in 50ths of second between each successive character. Example:

```
key$(1)="Key Speed 10,1"+Chr$(13) : Rem Save reasonable speed
Key Speed 10,10 : Rem Hold down a key for SLOW repeat
Key Speed 1,1 : Rem FAST repeat
Rem Press Left Amiga F1 to return your keyboard back to normal
```

CLEAR KEY *(Initialise keyboard buffer)*

CLEAR KEY

Whenever you enter a character from the keyboard, its Ascii code is placed in an area of memory known as the keyboard buffer. It is this buffer that is sampled by the INKEY$ function to get your key presses.

CLEAR KEY erases this buffer completely, and returns your keyboard to its original state. It's especially helpful at the start of a program, as the buffer may well be full of unwanted information. You can also call it immediately before a WAIT KEY command to ensure that the program waits for a fresh keypress before preceding. Example:

```
Clear key : Rem remove current key presses
Wait key : Rem Wait for a new key
```

PUT KEY *(Put a string into the keyboard buffer)*

PUT KEY a$

Loads a string of characters directly into the keyboard buffer. Carriage returns can be included using a CHR$(13) (RETURN) character.

The most common use of PUT KEY is to set up defaults for your INPUT routines. Here's a demonstration:

```
Do
  Put Key "No"
  Input "Another Game";A$
  If A$="No" Then Exit
Loop
```

The program above assigns "No" to the default INPUT string. Hitting the RETURN key will enter this value straight into the variable A$. Alternatively, you can edit the line using the normal cursor keys, and type in your own value for A$ as required.

Input/Output

INPUT *(Load a value from the user and put it a variable)*

INPUT provides you with a standard way of entering information into one or more variables. There are two possible formats for this instruction:

INPUT vars[;]

Enters a list of variables directly from the keyboard. *var* can contain any set of variables you like, separated by commas. A question mark will be automatically displayed at the current cursor position.

INPUT "Prompt";variable list[;]

Prints out the *prompt* string before entering your information. Note that you must always place a semi-colon between your text and the variable list. You are **not** allowed to use a comma for this purpose.

The optional semi-colon ";" at the end of your variable list specifies that the text cursor will not be affected by the INPUT instruction, and will retain its original position after the data has been entered.

When you execute one of these commands, Basic will wait for you to enter the required information from the keyboard. Each variable in your list must be matched by a single value from the user. These values must be of the same type as your original variables, and should be separated by commas. Here are some simple examples:

```
Input A
Print "Your number was";A
Input "Enter a floating point number";N#
```

```
Print “You entered”;N#
Input “What’s your name Human?”;name$;
Locate 23, : Print “Hello “;Name$
```

See INPUT# and LINE INPUT

LINE INPUT *(Input a list of variables separated by a Return)*

INPUT “Prompt”;variable list[;]
INPUT vars[;]

Line input is exactly the same as INPUT, except that it uses a Return instead of a comma to separate each value you enter from the keyboard. Example:

```
Line Input “Enter three numbers”;A,B,C
Print A,B,C
```

See INPUT, LINE INPUT#

19: Other commands

Like all versions of the Basic language, AMOS also includes a variety of mundane commands such as PRINT and DATA. As you will discover in this section, AMOS incorporates a number of exciting new twists to these instructions. So even the boring parts of AMOS Basic are quite interesting!

PRINT or ? *(Print a list of variables to the screen)*

PRINT items

The PRINT instruction displays some information on the screen, starting from the current cursor position.

The list of items can consist of any group of variables or constants you like, providing you don't exceed the maximum permitted line length (255).

Each element in your list must be separated by either a semi-colon ";" or a comma "," A semi-colon prints the data immediately after the previous value, whereas a comma first moves the cursor to the next TAB position on the screen.

Normally the cursor will be advanced downwards by a single line after each PRINT instruction. This can be suppressed by adding a separator after the print. As before, a semi-colon will preserve the cursor position after the operation, and a comma will place the cursor to the next TAB stop before proceeding.

```
Print "This Is The Story Of The Hitchhikers Guide To The Galaxy"
A=10 : B=20 : C$="Thirty": Print A,B;C$
Print 10,20*10,"Hel";
Print "lo"
```

See also USING, LPRINT and PRINT#

USING *(Formatted output)*

PRINT USING format$;variable list

The USING statement is used in conjunction with PRINT to provide fine control over the format of your printed output.

format$ specifies a list of characters which defines the way your variables will be displayed on the screen. Any normal text in this string will be printed directly, but if you include one of the characters ~#+-.;^ then one of a range of useful formatting operations will be performed.

~ (Shift+#) Formats a string variable. Every ~ is replaced by a single character from your output string, taken from left to right.

```
Print Using "This is a ~~~~~ demonstration of USING";"Small"
This is a small demonstration of USING
```

# Each hash character specifies a single digit to be printed out from your variable. Any unused digits in this list will be automatically replaced by spaces.

```
Print Using "####";314211
4211
```

+ Adds a plus sign to a number if it is positive, and a minus sign if it is negative.

```
Print Using "+##";10: Print Using "+##";-10
+10
-10
```

– Only includes a sign if the number is negative. Positive numbers are preceded by a space.

```
Print Using "-##";10 : Print Using "-##";-10
 10
-10
```

(Period) places a decimal point in the number, and centres it neatly on the screen.

```
Print Using "PI Is #.###";3.1415926
PI Is 3.141
```

; Centres a number but doesn't output a decimal point.

```
Print Using "PI Is #;###";PI#
PI Is 3 141
```

^ (Shift 6) prints out a number in exponential form.

```
Print Using " Here is a number ^";12345.678
Here is a number 1.2345678E5
```

REM or ‘ *(Remark)*

REM comment

The REM statement is used to add comments to your Basic program. Any text typed in after a REM statement will be completely ignored by AMOS Basic. Example:

```
Rem This program does absolutely nothing
' This is a comment
```

The standard REM statement can be used practically anywhere in your Basic program. But a quote ' mark can only be placed at the absolute beginning of one of your lines. Examples:

```
' A simple comment
Print "Hell" : Rem This is ok
Print "Goodbye" : ' This will generate an error
```

DATA *(Place a list of data items in a AMOS Basic program)*

DATA list of items

The DATA statement allows you to incorporate whole lists of useful information directly inside a Basic program. This data can be subsequently loaded into one or more variables using the READ instruction. Each variable in your list is separated by a single comma. Example:

```
Data 1,2,3,"Hello"
```

Unlike most other Basics, the AMOS version of this instruction also lets you include expressions as part of your data. So the following lines of code are all equally acceptable:

```
Data $FF50,$890
Data %1111111111111,%1101010101
Data A
Label: Data A+3/2.0-Sin(B)
Data "Hello"+"There"
```

It's important to realise that the "A" at LABEL will be input as the contents of variable A, and not the character A. The expression will be evaluated automatically during the READ operation using the latest values of A and B.

Also note that each DATA instruction must be the only statement on the current line. Anything after this command will be totally ignored! Data statements can be placed anywhere you like in your Basic program. However, any data you store inside and AMOS procedure will not be accessible from the main program. Each procedure can have its own individual set of DATA statements which are completely separate from the rest of your program. Here's a demonstration:

```
TEST : Read A$ : Print A$
Data "Program data"
Procedure Test
   Read B$ : Print B$
   Data "Procedure data"
End proc
```

See READ, RESTORE.

READ *(Read some data from a DATA statement into a variable)*

READ list of variables

READ loads some information stored in a DATA statement into a list of variables. READ uses a special marker to determine the location of the next piece of data to be entered. At the start of your program, the marker is moved to the first item of the first DATA statement. Once this item has been read, the marker is advanced so that it points to the next item in your list. As you might expect, the variables you read must be exactly the same type as the data held at the current position. Example:

```
T=10
Read A$,B,C,D$
Print A$,B,C,D$
Data "String",2,T*20+rnd(100),"AMOS"+"Basic"
```

See RESTORE, DATA.

RESTORE *(Set the current READ pointer)*

RESTORE Label
RESTORE LABEL$
RESTORE Line
RESTORE number

RESTORE changes the point at which a subsequent READ operation will expect to find the next DATA statement. Each AMOS procedure has its own individual data pointer. So any calls to this command will only apply to the **current** procedure!

label is a label which specifies the position of the first DATA statement to be read. This label name can be calculated as part of an expression. So the following Basic commands are all perfectly legal:

```
Restore L
Restore "L"+"A"+"B"+"E"+"L"
```

Similarly, *line* selects the line number of the next DATA statement. Like *label* it can be entered as an expression:

```
Restore 10
Restore TEST+2
```

By allowing you to jump at will through the DATA statements in your program,RESTORE lets you choose your information depending on the actions of the user. Each room of an adventure, for instance, could have its description stored in a list of simple DATA statements. To read this description you could use something like:

```
Restore ROOM*5+1000 : Rem Each ROOM has 5 data statements
Read DESC$ : Print DESC$
1000 Data "Description on Room 1"
1005 Data "Room 2 text"
1010 Data "Room 3"
:     :          :
```

Obviously, if a data statement does not exist at the line specified by RESTORE, an appropriate error message will be generated. Beware of trying to use this command inside a procedure. In order to work, your DATA statements MUST be within the current procedure.

See also READ, DATA

WAIT *(Wait in 50ths of a second)*

WAIT n

Suspends an AMOS Basic program for *n* 50ths of a second. Any functions which use interrupts, such as MOVE and MUSIC, will continue to work as normal during this period. Example:

```
wait 50
```

This waits for one second.

=TIMER= *(Count in 50ths of a second)*

v=TIMER
TIMER=v

TIMER is a reserved variable which is incremented by 1 every 50th of a second. It's commonly used to set the seed of the random number generator like so:

```
Randomize Timer
```

NOT *(Logical NOT operation)*

v=NOT(d)

This function changes every binary digit in a number from a 1 to a 0 and vice versa. Since True=-1 (%1111111111111) in binary and False=0, NOT(True)=False. Example:

```
Print Bin$(Not(%1010),4)
0101
If Not(True)=False Then Print "False"
```

TRUE *(Logical TRUE)*

v=TRUE

Whenever a test is made such as X>10, a value is produced. If the condition is true then this number is set to -1, otherwise it will be zero.

```
If -1 Then Print "Minus 1 Is TRUE"
If TRUE Then Print "and TRUE Is ";TRUE
```

See FALSE, NOT

FALSE *(Logical FALSE)*

v=FALSE

Returns a value of zero. This is used by all the conditional operations such as IF...THEN and REPEAT...UNTIL to represent FALSE.

```
Print FALSE
0
```

See TRUE.

20: Disc access

The AMOS disc commands give you total access to the Amiga's filing system. These can be exploited to create anything from a simple file reader to a fully fledged database.

AMOS is particularly impressive when it comes to selecting your files. There's a built-in file selector routine which allows you to choose your filenames from a fancy dialogue box. This is incredibly easy to use, and adds a truly professional touch to your Basic programs.

AMOS also includes the ability to read directories, delete files, or create folders directly from one of your programs. There's even a command which lets you check for the existence of a specific file on the disc!

Last, but not least, we've provided special support if you're upgrading from the original STOS Basic package. AMOS is fully compatible with the powerful CROSS DOS system from CONSULTRON. So if you've purchased this product, it will be very easy to import your existing STOS Basic programs straight into AMOS Basic.

Drives and volumes

As you know, the Amiga lets you label your discs in a number of different ways. If you're unfamiliar with the CLI, you may find some of these terms a little confusing. So I'll now provide you with a brief explanation of the various naming conventions.

Drives

Each drive connected to the Amiga is referred to by a standard three letter identification code. In order to distinguish this code from a normal file name, it's usually terminated by a colon character ":" when it's entered into your Basic instructions.

Floppy drives: Are assigned names in the following format:

```
Dfn:
```

n is a single digit which holds the number of your drive. The first floppy drive in your system (usually the internal drive) is known as Df0: then come drives Df1:, Df2: and Df3: if they're installed

Hard drives: These are specified using:

```
Dhn:
```

where *n* is the number of your hard drive.

Volumes

The Amiga also creates a separate VOLUME name for each individual disc. This label can be substituted for the drive name in any of your AMOS Basic commands. AMOS will automatically check each available drive for the required disc. If it can't be found, you'll get a *Drive not mounted* error.

Whenever you prepare a new disc from the Workbench, the disc will be assigned the name "Empty". To change this label from the Workbench, simply click on the RENAME option and enter your new name in the dialogue box provided. This name can be practically any string of characters you like, but it must be terminated with a colon character when it's used in your programs, just like the drive name. Here are some typical volume names for you to examine:

```
AMOS:
AMOS_DATA:
```

These could be used in the following way:

```
Load "AMOS:Sprite.Acc" : Rem Load the sprite editor from a disc called AMOS

Dir "AMOS_DATA:" : Rem Get the directory of AMOS_DATA:
```

Warning! If you create several discs with the same name or swap them around indiscriminately, the Amiga can easily get confused as to which disc you are actually referring to. In these circumstances, you'll need to enter the drive name instead. This will tell AMOS precisely where the required disc can be found on your system. Example:

```
Dir "Df0:"
```

You are strongly recommend to assign a different name to each and every disc you use. This takes no more than a couple of seconds from the Workbench but it does simplify things enormously.

Logical devices

Finally, there's also a set of objects known as logical devices.These are used by the Amiga's operating routines to determine the precise position of important system files such as device handlers or fonts. Each device is normally assigned to a specific directory on the current start-up disc. Here are some examples used by AMOS.

FONTS:	A directory containing the current fonts.
LIBS:	Holds a library file required by the AMOS Say command.

Try typing the following line from direct mode:

```
Dir "LIBS:"
```

Cross Dos

If you've bought the separate CROSS DOS package and installed it into memory, you'll also be able to access IBM or ST format discs within AMOS Basic. These discs are assigned names starting with the letters DI:

DIn:	(Where n is the number of your drive)

In order to convert your STOS programs to AMOS Basic, you'll need to save them in ASCII format using the FSAVE "*.ASC" option from STOS. Then insert the disc into an Amiga floppy drive that has been mounted by Cross-Dos as an IBM drive.

Note: Due to the differences between AMOS and STOS, many STOS programs will require modifying slightly before they will run under AMOS. If you want to make use of the special features of AMOS Basic, you may need to perform some quite drastic changes to certain programs. Despite this, it's still worthwhile taking the trouble to convert your existing STOS programs into AMOS format. The additional power of the Amiga's hardware can transform your STOS games out of all recognition!

Directory Changing

DIR *(Print out the directory of the current disc)*

DIR [PATH$] [/W]

Lists all the files on the current disc. If the optional path$ is specified, only the files which satisfy a certain set of conditions will be displayed. Any folders in the listing will be distinguished by a leading "*" character .

The listing can be halted at any time by pressing the spacebar. To resume, simply press the spacebar once again.

Note that if you change discs and try to get a directory listing, you may be presented with a *Device not mounted* error. This is because you've removed the current disc without informing AMOS Basic. The solution is to simply update the current directory name to the new disc using a line like: DIR$="Df0:" before calling DIR.

/W lists the files in two columns across the screen. This effectively doubles the number of files which can be displayed at any one time.

The path string consists of three main elements:

[Disc:][Directory/] Filter

Disc This is the name of the disc to be to examined. In order to stop your disc name being misinterpreted as just a normal file, you'll need to add a colon ":" at the end. Example:

```
Dir "AMOS:"
Dir "Df0:"
Dir "FONTS:"
```

Directory/ Holds the name of a single folder you wish to display. Examples:

```
Dir "AMOS:IFF/"
Dir "IFF/"
```

Filter Defines a set of conditions which should be satisfied for each file in your listing.

Normal text: Each character in your text should match exactly one character in the filename to be displayed. Example:

```
Dir “Music”
Music
```

** (asterix):* Matches any list of letters in your filenames up to the next control character. Example:

```
Dir “M*” : Rem List all files with names starting with the letter M
Music
Mercury
Malt
```

As a default, this option will ignore any files which include an *MS-DOS* type extension. So a file like: *Mad.Asc* on the disc would be not be listed.

. (period): Matches the extension of a filename. It’s commonly used in conjunction with the “*” command to list all files with a particular extension. Here are some examples:

```
Dir “*.*” : Rem List all files on the disc with an extension
Dir “*.Amos” : Rem List all AMOS files
Dir “Program.*” : Rem Displays all names starting with Program
```

?: Matches a single character at the current position. So **Dir “AM??”** will list files like:

```
AMOS
AMAL
```

However, it will ignore a name like “AMOS-BASIC” because it’s more than four characters long. Every letter in the filename has to match just a single character in the search string in order for the file to be displayed.

See LDIR

=DIR$= *(Change current directory)*

s$=DIR$
DIR$=s$

DIR$ contains the directory which will be used as the starting point for all future disc operations, such as loading and saving. It’s very similar to the CD command from the CLI, with the added advantage of allowing you to read the directory as well as just change it. Example:

```
Print Dir$ : Rem Print out current directory
Dir$=”AMOS:IFF/” : Rem Set directory to folder IFF on the disc AMOS
```

Like the CD function it replaces, all directories are assumed to be relative to the one you're currently using. Supposing you enter a line like:

```
Dir$=”MANUAL/”
```

This sets the current directory to MANUAL. If you now try to enter another folder, such as FONTS with DIR$=FONTS/” you'll get a *file not found* error. That's because AMOS will only search the current directory for your folder. Since FONTS cannot be found within the MANUAL folder, you'll get an error. To avoid this problem just include the disc name in you assignment like so:

```
Dir$=”AMOS:FONTS/”
```

or

```
Dir$=”Df0:FONTS/”
```

PARENT *(Sets the current path up one directory)*

The Amiga's filing system allows you to *nest* directories inside each other. This makes it very easy to organize your files according to a range of categories. Let's take a small example:

```
FOLDERA/
    FOLDERB/
        FOLDERC/
    FOLDERD/
```

The diagram above represents four separate folders on the disc. FOLDERA is stored in the main or *root* directory and contains FOLDERB and FOLDERD. In computing jargon, FOLDERA is known as the *parent* of these directories. Similarly, FOLDERB is the parent of FOLDERC.

As you might imagine, it's extremely easy to get totally lost when you are using these folders. The action of PARENT is to load the current directory with the parent of the present folder. By repeatedly using this command, you can quickly get back to your original root directory. Example:

```
Dir$=”\IFF”
Dir
Parent
Dir
```

SET DIR *(Set style used by DIR)*

SET DIR n[,filter$]

Sets the style of your directory listings. *n* is the number of characters ranging from 1-100 which will be displayed in each filename. Note that this setting has **no** effect on the actual length of your names. It only changes the way they will be listed on the screen.

filter is a list of pathnames which are to be **excluded** from your directory searches. All filenames that match this filter are completely ignored and will not be displayed as part of your directory. This can be used to suppress the annoying ".INFO" files which contain the icon definitions used by the Workbench.

Note that it's perfectly possible to ignore a whole list of filepaths at once. Simply terminate each name with a single "/" character. As a default, the filter is set to:

".INFO/*.INFO/*.*.INFO"

Example:

```
Set Dir 5 : Rem Only display the first five character in your names
Dir
Set Dir 30,"" : Rem No filter
Dir
```

Common disc operations

=DFREE *(Disc free space)*

f=DFREE

Returns the amount of free space remaining on the current disc, measured in bytes.

```
Print Dfree
```

MKDIR *(Create a folder)*

MKDIR f$

Creates a new folder on the disc with the name f$. Example:

```
Mkdir " Df0:TEST"
Dir
```

KILL *(Erase a file from the disc)*

KILL f$

Deletes the file f$ from the current disc. **Warning!** Anything you erase in this way will be destroyed!

RENAME *(Rename a file)*

RENAME old$ TO new$

Changes the name of a file. If a file already exists with the new name you have chosen an error will be generated.

Selecting a file

=FSEL$ *(Select a file)*

f$=FSEL$(path$[,default$][,title1$,title2$])

The FSEL$ function lets you choose your files directly from the disc, using the standard AMOS file selector.

path$ sets a search pattern which determines which files will be displayed in your listing.

After you've selected a file, FSEL$ will return either its full pathname, or an empty string "" if you selected QUIT.

default$ chooses a filename to be used as a default. This will be automatically selected if the user aborts by pressing Return.

title1$* and *title2$ are optional text strings which describe a title to be displayed at the top of your file selector. Example:

```
F$=Fsel$("*.IFF","","Load an IFF file")
If F$="" Then Edit : Rem Return to the editor if no file was chosen
Load IFF F$,0 : Rem Load File
```

Running an AMOS program from disc

RUN *(Execute an AMOS Basic program)*

RUN [file$]

Although it's easy enough to execute your AMOS programs straight from the editor, we've also included a separate RUN command. This version of the command without *file$* can only be used from direct mode.

But the RUN *file$* statement may also be placed inside a Basic program. This allows you to *chain* a list of programs together. Note that when you run a program in this way, the existing program will be removed from memory and any variables will be lost. Any data screens that have been created though, will remain intact, thus allowing intermediate loading screens to be displayed. Example:

```
Print "Executing Test"
Run "Hithere.AMOS " : Rem On AMOS Program Disc
Print "This Line Is Never Executed"
```

This command is fantastically useful, as it allows you to split any AMOS game into a number of levels which can be loaded separately from the disc. Each level can now be written as a completely independent program. So the only limit to the size of your games is the amount of storage space on the disc! You can therefore produce some massive games with this system!

See also PRUN

Checking for the existence of a file

=EXIST *(Check if specified file exists)*

flag=EXIST(f$)

EXIST checks the current directory for the file f$. If it is found, then a value of -1 (true) will be returned, otherwise 0 (false). This EXIST function is capable of checking for the existence of anything from a single file to an entire disc. Here are some examples:

```
Print Exist("This is a really silly name HAHA")
Print Exist("AMOS:") : Rem Is a disc called AMOS: available
Print Exist("Df1:") : Rem Has a second drive been connected
```

As you can see, EXIST will happily accept total gibberish without generating a single error. Providing the filename contains at least one character, EXIST will carry on regardless. Empty strings like "" should however, be tested separately. Example:

```
F$=Fsel$("*.IFF","","Load an IFF file")
If F$="" Then Edit : Rem Return to the editor if no file was chosen
If Exist(F$) Then Load Iff F$,0
```

=DIR FIRST$ *(Get first file in directory satisfying path name)*

file$=DIR FIRST$(path$)

Returns a string containing the name and length of the first file on the disc which satisfies the current search path$. When this function is called, the entire directory listing will be loaded into memory. You can now retrieve the name of the next file in the directory using a call to the DIR NEXT$ function.

```
Print Dir First$("*.*")
```

=DIR NEXT$ *(Get the next file satisfying current path)*

file$=DIR NEXT$

Returns the next filename in the directory listing created by a previous DIR FIRST$ command. After the last item has been read from this list, a string will be returned containing the empty string "". The entire directory array will now be erased and the memory it consumes will be released for the rest of your Basic program. Here's an example which prints out all the files in the current directory:

```
F$=Dir First$("*.*") : Rem Read directory and get the first item
While F$<>""
   Print F$ : Bell : Wait 30
   F$=Dir Next$ : Rem Get next file in list
Wend
```

A further demonstration of these commands can be found in **EXAMPLE 20.1** in the manual folder. This copies the contents of a folder from one disc to another.

Disc files

Files are just collections of information which have been grouped together in one place on the disc. Each file is assigned its own name which may contain anything from 1 to 255 characters.

Before you can use one of these files, you first need to initialize it using either the OPEN IN, OPEN OUT, or APPEND instructions. When you open a file, you assign it to a *channel* number ranging from one to ten. This number will be used in all future disc operations to identify the file you are currently working with.

The Commodore Amiga supports two different types of disc files: Sequential files and random access files.

Sequential files

Sequential files are the standard files which are used on the Amiga. The reason for their name is that they only allow you to read your information in the precise sequence it was originally created.

This means that if you wanted to change just one piece of the data in the middle of a sequential file, you would have to read in the whole file up to and including this value, and then write the entire file back to the disc.

AMOS Basic allows you to access sequential files for either writing or reading, but never for both at the same time.

```
Open Out 1,"file.seq"
Input "What is your name";N$
Print #1,N$
Close 1
```

This creates a file called FILE.SEQ containing your name. In order to read this information back from the file, type in the lines:

```
Open In 1,"file.seq"
Input #1,N$
Print "I remember your name. It is ";N$
Close 1
```

Notice how both these programs perform three separate operations.

- Open the file using either OPEN IN, OPEN OUT or APPEND
- Access the file with INPUT#, or PRINT#
- Close the file with CLOSE. Note that if you forget to do this, any changes to the file will be lost!

These three steps need to be completed in exactly this order, every time you access a sequential file.

OPEN OUT *(Open a file for output)*

OPEN OUT channel,n$

Opens a sequential file for writing. If this file already exists it will be erased. *channel* is a number between 1 and 10 and is used to identify your new file in your subsequent PRINT# commands.

n$ is the name of the file to be opened.

APPEND *(Add some information to the end of an existing file)*

APPEND channel,name$

APPEND opens a sequential file for output. If this file exists, the new data is added onto the end. This allows you to expand your files at any time once they've been defined.

OPEN IN *(Open a file for input)*

OPEN IN channel,f$

OPEN IN sets up a file for reading. If this file does not exist, it will be automatically created.

channel is a number ranging from 1 to 10 which is used by various INPUT instructions to refer to your open file.

CLOSE *(Close a file)*

CLOSE n

Closes file number *n*. **Warning!** If you forget to close a file after you have finished with it, any changes you have made to the file will be completely ignored.

PRINT # *(Print a list of variables to a file or device)*

PRINT#channel,variable list

This command is identical to the normal print instruction, but instead of displaying the information to the screen, it outputs it to a file or output device specified by the channel number. Here's an example:

```
Open Out 1,"Testfile"
Print #1,"Hello"
Close 1
```

As with PRINT you can abbreviate PRINT# to ?#.

INPUT # *(Input a list of variables from a file or device)*

INPUT #channel,variable list

INPUT# reads information from either a sequential file or a device such as the serial port. Like the standard INPUT command, it enters a list of values and loads them into a set of Basic variables. As always, each value in the list must be separated by a single comma. Additionally, every line of data also needs to be terminated by its own <Line feed> character. This is equivalent to the *Return* you pressed when you entered a line from the keyboard. Example:

```
Open In 1,"Testfile" : Rem Opens File Created In Previous Example
Input #1,A$
Print A$
Close 1
```

LINE INPUT # *(Input a list of variables not separated by a ",")*

LINE INPUT # has two possible formats:

LINE INPUT #channel,variable list

or

LINE INPUT #channel,separators,variable list.

This function is identical to INPUT#, except that it allows you to separate your list of data using any character you wish instead of the standard comma. If the separator is omitted, it's automatically set to the *Return* character.

When you are reading text, LINE INPUT# is always the preferred choice. That's because the commas found in normal English will be treated as a separator by the INPUT# command. This will confuse your program completely.

SET INPUT *(Set End of Line characters)*

SET INPUT c1,c2

Sets the End-of-Line characters which will be used to terminate a line of data. The Amiga expects a single <line feed> character at the end of each line, whereas most other computers (including the ST) require both a *Return* and <line feed>. So if you try importing your ST files via the serial cable, you'll end up with dozens of spurious *Return* characters in your files. Fortunately you can sidestep this problem using SET INPUT.

c1 and *c2* hold a pair of ASCII values which will be used for your separators. If you want to use a single character, simply load *c2* with a negative value such as minus one. Here's a couple of examples:

```
Set Input 10,-1 : Rem Standard Amiga format
Set Input 13,10 : Rem ST format
```

=INPUT$ *(Inputs a number of characters from a device)*

x$=INPUT$ (f,count)

Reads *count* characters from device or file number *f.*

=EOF *(Test for end of file)*

flag=EOF (channel)

EOF is a useful AMOS Basic function which tests to see if the end of a file has been reached at the current reading position. If it has, EOF returns a result of -1, otherwise 0.

LOF *(Length of open file)*

length=LOF(channel)

Returns the length of an open file. It makes no sense to use this function in conjunction with devices other than the disc.

POF *(Variable holding current position of file pointer)*

pos=POF(channel)

The POF function changes the current reading or writing position of an open file, for example:

```
Pof(1)=1000
```

This sets the read/write position to 1,000 characters past the start of the file. Oddly enough POF can be used in this way to provide a crude form of random access when using sequential files! The reason this works is simply that disc drives are inherently random and all sequential operations are effectively simulated using random access.

Random access files

Random access files are so called because you can access the information stored on the disc in any random order you like. In order to use these files you first need to understand a little bit of theory.

All random access files are composed of units called records, each with their own unique number. These records are in turn split up into a number of separate fields. Every field contains one individual piece of information. When you use sequential files, these fields can be any length you wish, as the file will only be read in one direction. Random access files, however, always require you to specify the maximum size of each of these fields in advance.

Supposing you wanted to produce a file containing a list of names and telephone numbers. In this case you could use the fields:

Field	Maximum length
SURNAME$	15
NAME$	15
CODE$	10
TEL$	10

You could now define these fields using a line like:

```
Field #1,15 as SURNAME$,15 as NAME$,10 as CODE$,10 as TEL$
```

It's important to realize that the strings specified by the FIELD instruction can also be used as normal string variables. This allows you to read and write information to any particular field. For example:

```
SURNAME$="HILL" : Rem Loads the surname into the field SURNAME$.
TEST$=SURNAME$ : Print TEST$
```

After you've loaded your record with information, you can write it onto the disc using the PUT command. Example:

```
Put 1,10
```

Loads data into record 10 of the file opened on channel 1.

Similarly, you can read a record using the GET instruction.

```
Get 1,10
```

OPEN RANDOM *(Open a channel to a random file)*

OPEN RANDOM channel,n$

Opens a random access file called *n$* on the current disc. When you're using this instruction, you should always define the record structure immediately afterwards using the FIELD$ command.

FIELD *(Define record structure)*

FIELD channel, length1 AS field1$, length2 AS field2$......

FIELD allows you to define a record which will be used for a random access file. This record can be up to 65535 bytes in length.

```
Field 1,15 as SURNAME$,15 as NAME$,10 as CODE$,10 as TEL$
```

PUT *(Output a record to a random access file)*

PUT channel,r

PUT moves a record from the Amiga's memory into record number *r* of a random access file. Before use, the contents of the new record should first be placed in the field strings defined by FIELD, using a statement such as:

```
SURNAME$="HILL"
```

Although you can write existing records in any order you like, you are not allowed to scatter records on the disc totally at random. This means that if you have just created a file you can't type in something like:

```
Put 1,1
Put 1,5
```

In this case, the PUT 1,5 instruction will generate an error, as there are no records in the file with numbers between 2 and 5. Record 2 must be the next *new* record in the file, then 3, 4... Example:

```
Open Random 1,"TELEPHONE"
Field 1,30 As NAME$, 30 As TEL$
INDEX=1
Do
  Input "Enter a name"; NAME$
  Exit If NAME$=""
  Input "Enter telephone number"; TEL$
  Put 1,Index : Inc INDEX
Loop
Close 1
```

GET *(Input a record from a random access file)*

GET channel,r

GET reads record number *r* stored in a random access file opened using OPEN. It then loads this record into the field strings created by FIELD. These strings can now be manipulated in the normal way.

Note that you can only use GET to retrieve records which are actually on the disc. If you try to grab a record number which does not exist then an error will be generated. Example:

```
Open Random 1,"TELEPHONE"
Field 1,30 As NAME$,  30 As TEL$
Do
  Input "Enter Record number"; INDEX
  Exit If INDEX=0
  Get 1,INDEX : Print NAME$ : Print TEL$
  Loop
Close 1
```

The printer

If you own a printer you'll be able to get full listings of all your AMOS programs. This is an invaluable aid during the debugging process and should be considered as an essential purchase for all serious programmers. The following commands give you easy access to the printer:

LLIST *(Print part or all of a program on a printer)*

LLIST

Lists the entire program straight to the printer. Try listing some of the Basic programs supplied on the DATA disc. These provide a perfect demonstration of the various programming techniques needed to write your own AMOS Basic games. Feel free to modify them as much as you like.

LPRINT *(Output a list of variables to the printer)*

LPRINT variable list

LPRINT is exactly the same as PRINT but sends your data to the printer instead of the screen. Example:

```
Lprint "Hello"
```

See PRINT, USING, PRINT#

LDIR *(List a directory to the printer)*

LDIR [PATH$] [/W]

Lists the directory of the current disc to the printer. See DIR for more details.

External devices

OPEN PORT *(Open a channel to an I/O port)*

OPEN PORT channel,"PAR:" (Opens a channel to the Parallel interface)
OPEN PORT channel,"SER:" (Open a channel to the RS232 port)
OPEN PORT channel,"PRT:" (Open a channel for the printer chosen in preferences)

OPEN PORT allows you to communicate with external devices such as the RS232 port. All the standard sequential file commands can be performed as normal, except for commands like LOF or POF which are obviously only relevant to disc operations. Example:

```
Open Port 1,"SER:"
For X=0 to 10
   Print #1,"AMOS BASIC"
Next X
Close 1
```

This program prints out ten lines of text on the device connected to the RS232 port. If your printer uses the parallel port change line 10 to:

```
Open Port 1,"PRT:"
```

Similarly you can input information from a device such as a modem with a line like:

```
Input #1,A$ : Print A$
```

When accessing these external devices all the normal input statements are available for your use, including INPUT$ and LINE INPUT.

=PORT *(Function to test if channel waiting)*

n=PORT(channel)

Tests to see if an input device is ready to send you some information. If the device is waiting for you to read it, a value of -1 (true) will be returned by this function, otherwise 0 (false).

21: Screen compaction

Although the Amiga's hardware is capable of displaying some stunning pictures, it's previously been almost impossible to achieve these effects in an actual game. The problem is simply one of memory. A single 320x255 64-colour screen consumes a massive 60k of RAM. That puts a real limit on the number of screens you can use in your games, especially if you wish the resulting programs to to run on an unexpanded A500.

One solution is to generate your screens with the AMOS Map definer. This is fine for the background screens in an arcade game, but it's not really suitable for the type of realistic images you'd find in an adventure.

Fortunately, there's a simple alternative in the form of the AMOS compaction instructions. These can quickly compress an entire screen into just a fraction of its original size. All the standard graphics modes are supported, including HAM. So there's nothing stopping you from incorporating some terrific pictures into your AMOS programs.

SPACK *(Screen compaction)*

SPACK s TO n [tx,ty,bx,by]

The SPACK command, (pronounced s-pack), packs screen *s* into memory bank *n*. Everything about the current image is saved, including its mode, screen size, offset, and display position. This allows you to recreate your screen in exactly it's original state.

s is the number of the screen which contains your image. *n* holds the number of a memory bank from 1-16. If this bank does not currently exist, it will be reserved for you automatically. Your new bank will be stored in FAST memory if it's available, and will be saved along with your Basic program. After you've called this function, the size of your screen can be found using LENGTH. Example:

```
F$=FSEL$(“*”,””,”Load A Picture”)
Load Iff F$,0
Spack 0 To 1:Rem Compress Screen Zero Into A File In Bank One
Print “The Length Of Your New Bank Is “;Length(1);” Bytes””
Wait Key
Screen Close 0
Unpack 1 To 0:Rem Recreate Compacted Screen
```

You don't, of course, have to compact an entire screen with this instruction. The optional parameters let you compress any rectangular section of the display you like.

tx,ty now hold the coordinates of the top left corner of this region. *bx,by* set the position of the bottom right corner of your image. All *x* coordinates are rounded to the nearest 8 pixel boundary.

Note that in order to achieve the maximum memory reduction, SPACK will attempt to compact your image using several different strategies. It will then compress your image using the method which consumes the least amount of memory. One side effect of this efficiency, is that it usually takes around 6 seconds to compress one of your images. This is hardly a disadvantage however, as normally the compaction is only required when you are writing your programs.

Since each image can be unpacked on the screen in less than a second, there's no risk of interference with the speed of your games. If speed is of the essence though, you may wish to use the CBLOCK system instead. See the section on Background Graphics for more details.

Incidentally, if you compare the compacted size of your files with their original length on the disc, you may be mislead into underestimating the size of the memory reduction. It's important to realize that the vast majority of these files have ALREADY BEEN COMPRESSED using the standard IFF compaction routines. So it's rather surprising that AMOS can reduce them by a further 20%!

Compacted screens are perfect for the titles and hi-score tables required in an arcade game, as they allow you to introduce snazzy screen effects without consuming enormous quantities of memory. They can also be incorporated directly into RPGs and Adventures.

PACK *(pack a screen)*

PACK s TO n [tx,ty,bx,by]

PACK compresses screen *s* into bank number *n.* Unlike the previous SPACK command, only the image data is compressed. So your compacted screen must always be unpacked directly into an existing screen.

Because of the way images are decompressed, there will be a noticeable shimmer effect unless you've previously double buffered your screens. Try to avoid using PACK with single buffered screens. It's much more sensible to call the SPACK system for this purpose.

If the optional coordinates are included, only a section of the image will be compressed. *tx,ty* specify the position of the top left hand corner of this region. *bx,by* hold the coordinates of the bottom corner of your image. As before, all *x* coordinates are rounded down to the nearest 8 pixel boundary.

Since PACK is fully compatible with the standard Autoback system, it's easy to combine compacted images with moving screens. If you are using Autoback 2 mode, you'll even be able to unpack your images BEHIND existing bobs. It's therefore possible to exploit this instruction in conjunction with SCREEN OFFSET to create fantastic scrolling backgrounds for your games.

UNPACK *(Unpack a compacted screen)*

Decompresses a screen which has been previously compacted using the SPACK or PACK instructions. Each compaction routine has its own specific form of the UNPACK command.

SPACK

UNPACK b TO s

Opens screen *s* and restores the compacted screen into bank *b.* If this screen already exists, it will be completely replaced by the new image. Once the screen has been unpacked, it will be neatly flicked into view.

PACK

PACKed screens can be unpacked using two separate instructions. These take an image from a memory bank, and load it straight into an existing screen. **Warning!** The destination screen **must** be in exactly the same format as your compacted picture, otherwise you'll get an *illegal function call* error.

When you unpack your screens, you may notice that the effect is slightly messy. That's because the PACK command is only really intended for use with the double buffering system. Providing your screen is double buffered, you'll get a delightfully smooth result.

UNPACK b

Unpacks the screen at its original position

UNPACK b,x,y

Redraws your image starting at coordinates *x,y*. If the new image does not fit into the current screen, you'll get an appropriate error message.

22: Machine code

AMOS Basic includes a range of commands which provide the advanced programmer with total access to the inner workings of the Amiga's hardware. None of these instructions are required for general programming purposes, of course.

AMOS Basic has been carefully designed to allow you to control all the Amiga's features using simple, easy to understand Basic instructions. So there's no real need to hack around in memory in order to create your games!

Number conversion

=HEX$ *(Convert number to hexadecimal)*

```
h$=HEX$(v)
h$=HEX$(v,n)
```

HEX$ converts the integer *v* into hexadecimal notation (base 16). It returns a sequence of *n* hexadecimal characters in the string *h$*. Examples:

```
Print Hex$(Colour(1),3)
$A40
Print Hex$(65536)
$10000
Print Hex$(65536,8)
$00010000
```

=BIN$ *(Convert number to binary string)*

```
b$=BIN$(v)
b$=BIN$(v,n)
```

Converts a number into binary notation (base 2). As with HEX$, you can choose whether to output all the digits in the number, or only a few. Example:

```
Print Bin$(255)
%11111111
Print Bin$(255,16)
%0000000011111111
```

n specifies the number of binary digits which will be returned in *b$*. Each number can include anything from 1 to 32 digits (1<=n<=31).

Memory manipulation

=PEEK *(Get byte at address)*

```
v=PEEK(address)
```

Returns the 8-bit byte stored at *address*.

```
Load "AMOS_DATA:Sprites/Monkey.abk"
Print Peek(Start(1))
For S=8 To 1 Step -1
   C=peek(start(1)-S) : Print CHR$(c); : Rem So that's where the name is
Next S
```

POKE *(Change byte at address)*

POKE address,v

Copies the number *v* into *address*. *v* must always lie in the range 0-255.

You can use this function to change the contents of any part of the Amiga's memory you wish. But be warned that POKE can be very dangerous. If you poke around indiscriminately, you will almost certainly crash the Amiga completely. Example:

```
Load "AMOS_DATA:Sprites/Octopus.abk"
Poke (Start(1)-5),asc("o") : Rem Change one letter in the bank name
Listbank
```

=DEEK *(Get word at address)*

v=DEEK(address)

Reads the two-byte word found at *address*. *address* MUST be even or an address error will occur.

DOKE *(Change word at address)*

DOKE address,value

DOKE loads a two-byte number between 0 and 65535 into the memory location at *address*. In knowledgeable hands this function can be very useful. Since even the best of us make mistakes however, you should always save a copy of your programs to the disc before attempting to use this function in a new routine. Example:

```
Doke Phybase(1)+1000,65535
```

=LEEK *(Read a long word at address)*

v=LEEK(address)

Returns the four-byte long word stored at *address*. Like DEEK, the address used with this function must always be EVEN. If bit 31 of the return value is set to a 1, *v* will be displayed as a negative number. This isn't a bug. It's just a side effect of the way AMOS deals with numbers. Example:

```
Print Leek(0)
```

LOKE *(Change long word at address)*

LOKE address,n

LOKE copies the four-byte number *n* into *address*. Example:

```
Loke phybase(1)+10000,$FFFFFFFF
```

Indiscriminate use of this function can lead to the Amiga crashing horribly, so take care.

=VARPTR *(Get address of a variable)*

address=VARPTR(variable)

Returns the address in memory of a Basic variable. Each type of variable is stored using its own individual format:

Integers: VARPTR finds the address of the four bytes containing the contents of your variable. Example:

```
A=0
Loke Varptr(A),1000
Print A
1000
```

Floating point: VARPTR returns the location of four bytes which hold the value of the variable in the IEEE single precision format.
Strings: The VARPTR address points to the first character of the string. Since AMOS Basic does not end its strings with a CHR$(0), you must obtain the length of the string using something like: DEEK(VARPTR(A$)-2), where *A$* is the name of your variable. You could also use LEN(A$) of course.

COPY *(Copy a memory block)*

COPY start,finish TO destination

This command is used to rapidly move large sections of the Amiga's memory from one place to another. *start* and *finish* are the addresses of the first and last bytes of your data respectively. *destination* points to a memory area which will be loaded with your new data.

All these addresses **MUST** be even, or you'll get an address error. Example:

```
Reserve As Data 10,1000
A$="Hi There"
Copy Varptr(A$),Varptr(A$)+Len(A$) To Start(10)
For P=0 To Len(A$)
   Print Chr$(Peek(Start(10)+P));
Next P
```

FILL *(Fill memory block with a longword)*

FILL start TO finish,pattern

Fills a selected region of memory with the four bytes held in *pattern.*

start and *finish* determine the position and size of the block which is to be filled. **Warning!** These addresses MUST be even!

pattern is a long word containing a four byte fill pattern. This will be copied into each group of four memory locations between *start* and *finish.* Example:

```
Reserve As Data 10,1000
Rem create a string of four characters (4 bytes=1 Longword)
A$="AMOS"
Rem Fill Bank with string
Fill Start(10) To Start(10)+Length(10),Leek(Varptr(A$))
For P=0 To Lenght(10)
   Print Chr$(Peek(Start(10)+P));
Next P
```

=HUNT *(Find a string in memory)*

f=HUNT(start TO finish, s$)

Searches through the Amiga's memory for the sequence of characters held in *s$. start* is the address of the first byte in memory to be searched, and *finish* is the address of the last.

On completion, *f* will hold either 0 (if the string in a$ was not found) or the location of *f$.*

Bitwise operations

ROL *(Rotate left)*

ROL.B n,v
ROL.W n,v
ROL.L n,v

ROL is a Basic version of the ROL instruction found in 68000 assembly language. The effect is to take the binary representation of a number in *v*, and rotate it left by exactly *n* places.

If *v* is a single variable, then the number to be rotated is taken directly from *v*. But if *v* is an expression, then it's treated as the **address** of your number instead. Here's an example:

The number 136 is represented in binary by:

```
%10001000
```

Type in:

```
V=136
Rol.B 1,V
Print Bin$(V,8)
%00010001
```

Notice how the first bit in the number has magically re-appeared from the back. Now for an example of the address system.

```
Reserve As Work 10,1000
Poke Start(10),%00001111
Rol.B 1,Start(10)
Print Bin$(Peek(Start(10)),8)
%00011110
```

This will give the number 17 or binary %00010001

As you can see, the entire number has been shifted to the left, with the highest 1 being rotated into the lowest position. The reason for the ".B", is to instruct AMOS to treat this number as an 8-bit byte. You can also specify the sizes ".W" (word) and ".L" (long word). Examples:

```
Rem Note the . between Rol and L
A=1
Rol.L 1,A
Print A
2

Rem Take note of the . between Rol and W
A=32768
Print BIN$(A)
Rol.W 2,A
Print A
2
```

ROL can be used as a very quick way of multiplying a positive number by a power of 2.

ROR *(Rotate right)*

ROR.B n,v
ROR.W n,v
ROR.L n,v

This is very similar to ROL but rotates the number in the opposite direction. As before, *v* may be either a simple variable or an expression. If it's an expression, ROL will rotate the number stored at the resulting **address**. Example:

```
A=8
Ror 1,A
Print A
4
```

ROR is capable of dividing any positive value by a power of two. The resulting calculation will be performed much faster than the equivalent "/" operation.

=BTST *(Test a bit)*

b=BTST(n.v)

Tests the binary digit at position *n* in the variable *v*. If *v* is an expression, it will be used as the address of the bit which is to be checked. In this case *n* will be automatically ANDed with 7 before proceeding.

After BTST has been called, *b* will be loaded with -1 (true) if the bit at position *n* is set to 1, otherwise it will be 0 (false). Example:

```
B=%1010
Print Btst (3,B)
-1
Print Btst (2,B)
0
```

See also BCHG, BCLR, BSET

BSET *(Set a bit to 1)*

BSET n,v

Sets the bit at position *n* to 1 in the variable *v*. If you substitute and expression for this variable, it will be treated as the address of a value in the Amiga's memory.Example:

```
A=0
Bset 8,A
Print A
256
```

BCHG *(Change a bit)*

BCHG n,v

Changes bit number *n* in the variable *v*. If this bit is currently 1 then the new value will be a zero, and vice versa. As usual, if *v* is replaced by an expression, then the result will taken as an address. Example:

```
A=0
Bchg 1,A
Print A
2

Bchg 1,A
Print A
0
```

BCLR *(Clear a bit)*

BCLR n,v

Clears bit number *n* in variable *v* by setting it to zero. Like all the Bitwise operations, if *v* is an expression, then it will be used an the location of your data in memory. Example:

```
A=128
Bclr 7,A
Print A
0
```

Using assembly language

AMOS Basic includes special facilities which allow you to combine assembly language routines with your Basic programs. It's worth emphasising that, because of the sheer power of the AMOS system, machine code is only rarely useful. We've added these features solely for existing assembly language programmers who may wish to optimise their Basic programs with the occasional bit of machine code.

Be warned! These commands are extremely dangerous, and can easily crash your Amiga if they're used recklessly. Unless you're fully at home with the intricacies of the Amiga's hardware, you are strongly recommended to avoid these functions completely!

PLOAD *(Reserve a memory bank for some machine code)*

PLOAD "filename",bank

Reserves a memory bank and loads it with a machine-code program from the disc.

bank is the number of a memory bank which is to be reserved for your program. If it's negative, then the bank will be calculated using the absolute value of this number, and the required memory area will be allocated from CHIP memory.

Once you've loaded a program in this way, you can save it on the disc as an normal ".ABK" file. Since the banks created by this function are permanent, it will also be saved directly with your AMOS programs.

Your program must consist of a machine code file in the standard Amiga format. In practice, it can contain practically anything you like, with the following restrictions:

- The code **must** be totally relocatable. as it will be positioned at the first free memory location which is available.

- Only the **code** chunk of your program will be loaded.

- The program must be terminated by a single RTS instruction.

CALL *(Call a machine-code program)*

CALL address[,params]
CALL bank[,params]

Executes an assembly language program held in the Amiga's memory.

address can be either the absolute location of your code, or the number of an AMOS memory bank which has been previously created with PLOAD.

On entry to your program, registers D0 to D7 and A0 to A2 will be loaded from the values stored in the DREG and AREG arrays. Your assembly language program can now change any 68000 registers it likes. At the start the routine, register A3 will point to the optional parameter list, and A5 will contain the address of the main AMOS data area.When your program's finished, you can return to Basic with just a simple RTS instruction.

params is a list of parameters which will be pushed onto the A3 stack by the CALL command. These parameters need to be removed in REVERSE order. So the last value you entered into the instruction will be the first on to the stack. Depending on the type of your parameters, the values referenced by A3 will be in one of the following three formats:

Integer: Holds a long word containing a normal AMOS integer.

Floating point: Contains a floating point number in IEEE single precision format.

String: Stores the address of the string. All strings start with a single word containing their length.

WARNING! Never poke directly into a string! When a string is initialised to a constant, the string address will point to the original assignment statement inside the current program listing! So if you change this value, you'll affect your original source code. This is obviously extremely unwise, and should be avoided.

For a demonstration of PLOAD and CALL see **EXAMPLE 21.1** in the MANUAL folder.

=AREG= *(Variable used to pass information to the 68000's address registers)*

a=AREG(r)
AREG(r)=a

AREG is an array of six PSEUDO variables which are used to hold a copy of the first six of the 68000's address registers. *r* can range from 0 to 6 and indicates the number of the address register which is to be affected.

Whenever the CALL command is executed, the contents of this array are loaded automatically into address registers A0 to A2. At the end of the function, they are then saved back with any new information which has been placed in the appropriate registers.

=DREG= *(Variable used to pass information to the 68000's data registers)*

d=DREG(r)
DREG(r)=d

This is an array of eight integers which holds a copy of the contents of the 68000 data registers. *r* refers to the register number and can range from 0 to 7 for D1 to D7 respectively.

Accessing the system libraries

AMOS also allows to you call up most of the Amiga's internal system libraries directly from the ROM. These aren't particularly useful, since all the really interesting calls have already been built into AMOS!

Don't use these routines unless you know precisely what you are doing. The Amiga is notoriously difficult to program, and it's all too easy to crash the system and generate the infamous GURU error by mistake.

=DOSCALL *(DOS library)*

r=DOSCALL(function)

Executes a function directly from the DOS library. *function* is the offset to the appropriate function. See the Amiga ROM Kernel Manuals for more details.

Before using this function, you'll need to set some of the control registers in D0 to D7 and A0 to A2 using the AREG and DREG functions. When the routine exits back to Basic, the contents of D0 are returned in *r*. **Note:**The return values will not be loaded into DREG and AREG.

=EXECALL *(EXEC Library)*

r=EXECALL(function)

Performs a call to the Amiga's EXEC library. On entry, D0 to D7 and A0 to A2 are loaded with the control settings from the DREG and AREG arrays. *r* is returned holding the contents of D0.

=GFXCALL *(Graphics library)*

r=GFXCALL(function)

Calls a routine from the graphics library using the control values stored in the DREG and AREG arrays.

function is the offset to the relevant function. *r* is the result of the operation returned in D0.

=INTCALL *(Intuition library)*

r=INTCALL(function)

Executes a command from the Intuition library. As usual the control values are loaded from DREG and AREG arrays, and *r* holds the result of the call.

Since AMOS doesn't use the standard Intuition routine, this function is especially dangerous. Only call it if you are already familiar with the Amiga's Intuition library.

Inside AMOS Basic

In order to provide full access to the inner workings of the AMOS system for developers, we've included several "hooks" into the various data areas. These are not intended for the casual programmer, but they do enable advanced users to create their own AMOS utilities.

=SCREEN BASE *(Get screen table)*

```
table=SCREEN BASE
```

Returns the base address of the internal table used to hold the number and position of your AMOS screens. See **EXAMPLE 21.2** for a simple demonstration.

=SPRITE BASE *(Get sprite table)*

```
table=SPRITE BASE (n)
```

Provides the address of the internal data list for sprite *n*. If this sprite does not exist, then the address of the *table* will be zero.

Negative values for *n* return the address of the optional MASK associated with your sprite. *table* will now contain one of three possible values, depending on the status of this mask:

table<0	Indicates that there's no mask for this sprite at all.
table=0	Sprite *n* does have a mask, but it hasn't yet been generated by the system.
table>0	This is the address of the Mask in memory. The first long word of this area holds the length of the mask, and the next is followed by the actual definition.

See **EXAMPLE 21.3** from the MANUAL folder.

=ICON BASE *(Get icon base)*

```
table=ICON BASE(n)
```

Returns the address for icon *n*. The format of this information is exactly the same as the previous SPRITE BASE function.

Command index

ABS 155
ACOS 112
ADD 39
AMAL 186
AMAL FREEZE 193
AMAL ON/OFF 193
AMALERR 195
AMPLAY 193
AMREG 193
ANIM 205
ANIM FREEZE 206
ANIM ON/OFF 206
APPEAR 136
APPEND 269
AREG 286
ASC 58
AT 91
ATAN 113
AUTO VIEW ON/OFF 121
AUTOBACK 158
BANK TO MENU 221
BAR 68
BCHG 284
BCLR 285
BELL 234
BGRAB 31
BIN$ 279
BLOAD 51
BOB 155
BOB CLEAR 161
BOB COL 170
BOB DRAW 161
BOB OFF 163
BOB UPDATE 161
BOBSPRITE COL 170
BOOM 233
BORDER 101
BORDER$ 98
BOX 65
BREAK ON/OFF 83
BSAVE 51
BSET 284
BTST 284
CALL 285
CDOWN 93
CDOWN$ 93
CENTRE 96
CHANAN 195
CHANGE MOUSE 165
CHANMV 195
CHANNEL 197
CHOICE 214,216
CHR$ 58
CIRCLE 66
CLEAR KEY 251
CLEFT 94
CLEFT$ 94
CLINE 96
CLIP 71
CLOSE 269
CLOSE EDITOR 30
CLOSE WORKBENCH 30
CLS 131
CLW 103
CMOVE 90
COL 170
COLOUR 62
COP LOGIC 144
COP MOVE 143
COP MOVEL 143
COP RESET 144
COP WAIT 143
COPPER OFF 143
COPPER ON 143
COPY 281
COS 112
CRIGHT 94
CRIGHT$ 94
CUP 93
CUP$ 93
CURS ON/OFF 95
CURS PEN 96
DATA 256
DEC 39
DEEK 280
DEF FN 118

DEF SCROLL 133
DEFAULT 121
DEFAULT PALETTE 131
DEGREE 111
DEL BLOCK 210
DEL CBLOCK 211
DEL ICON 209
DEL WAVE 244
DFREE 265
DIM 36
DIR 262
DIR FIRST$ 267
DIR NEXT$ 267
DIR$ 263
DIRECT 80
DO...LOOP 79
DOKE 280
DOSCALL 287
DOUBLE BUFFER 156
DRAW 65
DREG 286
DUAL PLAYFIELD 127
DUAL PRIORITY 128
EDIT 80
ELLIPSE 66
END 81
EOF 271
ERASE 50
ERRN 86
ERROR 86
EVERY n GOSUB 82
EVERY n PROC 82
EVERY ON/OFF 82
EXECALL 287
EXIST 267
EXIT 79
EXIT IF 80
EXP 114
FADE 137
FALSE 259
FIELD 272
FILL 282
FIRE 169
FIX 117
FLASH 138
FLIP$ 57
FN 118
FONT$ 107
FOR...NEXT 77
FREE 53
FSEL$ 266
GET 273
GET BLOCK 209
GET BOB 163
GET CBLOCK 211
GET DISC FONTS 107
GET FONTS 106
GET ICON 208
GET ICON PALETTE 208
GET PALETTE 131
GET ROM FONTS 107
GET SPRITE 153
GET SPRITE PALETTE 151
GFXCALL 287
GLOBAL 46
GOSUB 74
GOTO 73
GR LOCATE 63
GR WRITING 70
HCOS 113
HEX$ 279
HIDE 165
HOME 92
HOT SPOT 171
HSCROLL 98
HSIN 113
HSLIDER 104
HTAN 113
HUNT 282
HZONE 173
I BOB 162
I SPRITE 154
ICON BASE 288
IF...THEN...[ELSE] 75
IF...[ELSE]...ENDIF 76
INC 39
INK 61
INKEY$ 249
INPUT 252
INPUT # 269
INPUT$ 271
INPUT$() 250
INSTR 56
INT 115
INTCALL 287
INVERSE ON/OFF 88

JDOWN 169
JLEFT 168
JOY 168
JRIGHT 168
JUP 168
KEY SHIFT 250
KEY SPEED 251
KEY STATE 250
KILL 265
LDIR 274
LED 248
LEEK 280
LEFT$ 54
LEN 58
LENGTH 50
LIMIT BOB 162
LIMIT MOUSE 167
LINE INPUT 253
LINE INPUT # 270
LISTBANK 49
LLIST 274
LN 114
LOAD 51
LOAD IFF 124
LOCATE 90
LOF 271
LOG 114
LOGBASE 135
LOGIC 136
LOKE 281
LOWER$ 57
LPRINT 274
MAKE ICON MASK 209
MAKE MASK 172
MATCH 60
MAX 116
MEMORIZE X/Y 95
MENU$ 212,215
MENU ACTIVE 229
MENU BAR 228
MENU BASE 231
MENU CALC 222
MENU CALLED 226
MENU DEL 221
MENU INACTIVE 229
MENU ITEM MOVABLE 230
MENU ITEM STATIC 230
MENU KEY 219
MENU LINE 227
MENU LINKED 231
MENU MOUSE 232
MENU MOVABLE 229
MENU ON/OFF 213,220,221
MENU ONCE 227
MENU SEPARATE 231
MENU STATIC 230
MENU TLINE 228
MENU TO BANK 221
MENU X 231
MENU Y 231
MID$ 54
MIN 117
MKDIR 265
MOUSE CLICK 166
MOUSE KEY 166
MOUSE ZONE 173
MOVE FREEZE 205
MOVE ON/OFF 205
MOVE X 203
MOVE Y 204
MOVON 205
MUSIC 238
MUSIC OFF 238
MUSIC STOP 238
MVOLUME 239
NO MASK 158
NOISE 244
NOT 258
ON ERROR GOTO 83
ON ERROR PROC 84
ON MENU DEL 218
ON MENU GOSUB 218
ON MENU GOTO 218
ON MENU ON/OFF 218
ON MENU PROC 217
ON...GOSUB 81
ON...GOTO 81
ON...PROC 81
OPEN IN 269
OPEN OUT 269
OPEN PORT 274
OPEN RANDOM 272
PACK 277
PAINT 67
PALETTE 63
PAPER 87

PAPER$() 88
PARAM 46
PARENT 264
PASTE BOB 163
PASTE ICON 207
PEEK 279
PEN 87
PEN$ 87
PHYBASE 135
PHYSIC 135
PI# 111
PLAY 240
PLOAD 285
PLOT 64
POF 271
POINT 64
POKE 280
POLYGON 68
POLYLINE 65
POP 75
POP PROC 47
PORT 275
PRG FIRST$ 31
PRG NEXT$ 31
PRINT # 269
PRINT or ? 254
PRIORITY ON/OFF 174
PROCEDURE 42
PRUN 31
PSEL$ 32
PUT 273
PUT BLOCK 210
PUT BOB 163
PUT CBLOCK 211
PUT KEY 252
RADIAN 111
RAIN 141
RAINBOW 141
RANDOMIZE 116
READ 256
REM or ‘ 255
REMEMBER X/Y 96
RENAME 265
REPEAT$ 97
REPEAT...UNTIL 78
RESERVE 49
RESERVE ZONE 172
RESET ZONE 174
RESTORE 257
RESUME 85
RETURN 74
RIGHT$ 54
RND 115
ROL 282
ROR 283
RUN 266
SAM BANK 236
SAM LOOP 237
SAM PLAY 235
SAM RAW 236
SAMPLE 244
SAVE 51
SAVE IFF 124
SAY 247
SCAN$ 29
SCANCODE 249
SCIN 130
SCREEN 129
SCREEN BASE 288
SCREEN CLONE 127
SCREEN CLOSE 121
SCREEN COLOUR 130
SCREEN COPY 132
SCREEN DISPLAY 125
SCREEN HEIGHT 130
SCREEN HIDE 129
SCREEN OFFSET 126
SCREEN OPEN 119
SCREEN SHOW 130
SCREEN SWAP 134
SCREEN TO BACK 129
SCREEN TO FRONT 129
SCREEN WIDTH 130
SCROLL 134
SET BOB 157
SET BUFFER 53
SET CURS 95
SET DIR 264
SET ENVEL 245
SET FONT 108
SET INPUT 270
SET LINE 67
SET MENU 232
SET PAINT 70
SET PATTERN 69
SET RAINBOW 139

SET SLIDER 104
SET SPRITE BUFFER 151
SET TAB 97
SET TALK 247
SET TEMPRAS 71
SET TEXT 108
SET WAVE 241
SET ZONE 172
SGN 115
SHADE ON/OFF 88
SHARED 45
SHIFT DOWN 139
SHIFT OFF 139
SHIFT UP 138
SHOOT 233
SHOW 165
SIN 111
SORT 59
SPACE$ 57
SPACK 276
SPRITE 145
SPRITE BASE 288
SPRITE COL 169
SPRITE OFF 152
SPRITEBOB COL 170
SPRITE UPDATE 152
SQR 114
START 50
STR$ 59
STRING$ 58
SWAP 117
SYNCHRO 202
TAB$ 97
TAN 112
TEMPO 239
TEXT 106
TEXT BASE 109
TEXT LENGTH 109
TEXT STYLE 109
TIMER 258
TITLE BOTTOM 101
TITLE TOP 101
TRUE 258
UNDER ON/OFF 89
UNPACK 277
UPDATE 175
UPDATE EVERY 201
UPPER$ 57
USING 254
VAL 59
VARPTR 281
VIEW 121
VOICE 239
VOLUME 235
VSCROLL 99
VSLIDER 104
VUMETER 240
WAIT 258
WAIT KEY 259
WAIT VBL 136
WAVE 243
WHILE...WEND 78
WIND CLOSE 102
WIND MOVE 102
WIND SIZE 103
WINDON 102
WINDOPEN 99
WINDOW 102
WINDOW FONT 100
WINDSAVE 100
WRITING 89
X BOB 162
X GRAPHIC 92
X HARD 154
X MOUSE 167
X SCREEN 153
X SPRITE 152
XCURS 94
XGR 64
XTEXT 91
Y BOB 162
Y GRAPHIC 92
Y HARD 154
Y MOUSE 167
Y SCREEN 153
Y SPRITE 153
YCURS 95
YGR 64
YTEXT 92
ZONE 173
ZONE$ 98
ZOOM 142

MANDARIN
SOFTWARE